2

RADC-TR-89-339
Final Technical Report
February 1990

DISTRIBUTED OBJECT ORIENTED PROGRAMMING

The MITRE Corporation

E. H. Bensley, T.J. Brando, J.C. Fohlin, M.J. Prelle, A.M. Wollrath

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-339 has been reviewed and is approved for publication.

APPROVED: Charles B Schultz

CHARLES B. SCHULTZ
Project Engineer

APPROVED: Raymond P Urtz

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
Distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-89-339

6a. NAME OF PERFORMING ORGANIZATION
The MITRE Corporation

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COTD)

6c. ADDRESS *(City, State, and ZIP Code)*
Burlington Road
Bedford MA 01730

7b. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Rome Air Development Center

8b. OFFICE SYMBOL *(If applicable)*
COTD

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F19628-86-C-0001

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
62702F	MOIE	78	50

11. TITLE *(Include Security Classification)*
DISTRIBUTED OBJECT ORIENTED PROGRAMMING

12. PERSONAL AUTHOR(S)
E. H. Bensley, T. J. Brando, J. C. Fohlin, M. J. Prelle, A. M. Wollrath

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM Oct 87 TO Sep 88

14. DATE OF REPORT *(Year, Month, Day)*
February 1990

15. PAGE COUNT
72

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Concurrent modeling
Object oriented programming
Fault tolerant resource management

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This paper describes a concurrent model of execution for object oriented programming. In addition, it describes a computation manager that implements this model and a compiler that translates a subset of Common Lisp with the Flavors object oriented extension to code executable by the computation manager. Finally, fault tolerant resource management algorithms and a simulation of a multiprocessor system that embodies these algorithms are described.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Charles B. Schultz

22b. TELEPHONE *(Include Area Code)*
(315) 330-3623

22c. OFFICE SYMBOL
RADC (COTD)

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

SECTION PAGE

LIST OF FIGURES

FIGURE PAGE

SECTION 1

INTRODUCTION

1.1 OBJECTIVE

The objective of this project is to investigate the implementation of object oriented programming in a large distributed memory Multiple Instruction Multiple Data stream (MIMD) processing system. The amount of inherent parallelism in a program depends on the number of independent computational units that can be executed concurrently. To increase parallelism, we must strive to increase the number of these units.

An object oriented program consists of a collection of self-contained program units, called *objects*, that communicate with each other by sending messages. An object encapsulates both the data and the code necessary to manipulate the data. Thus, objects represent the independent computational units that can enable parallelism within a program. By distributing the objects over multiple processors, parallelism can be exploited to speed up program execution. Most importantly, this encapsulation is a natural part of the object oriented model of computation. Therefore, object oriented programming can provide the programmer with good conceptual tools to divide his software application into elements that can be readily distributed among many processors.

1.2 BACKGROUND

MITRE -- through its Future Generation Computer Architectures (FGCA) program -- has conducted research in parallel computing since 1983 [Harris85, Brando86, Brando87, Prelle87]. Our original goals were to investigate both hardware and software techniques for realizing speedup through parallelism. The kinds of programs we consider are completely general purpose. This means we must be able to support dynamic memory allocation, automatic memory deallocation, and dynamic communication patterns among heterogeneous asynchronous program elements. Our research has evolved so that it is currently directed toward operating systems for massive distributed-memory MIMDs (multiple instruction stream, multiple data stream machines) running general-purpose programs.

Scalability and reliability are central to our research. By scalability, we mean a system can be expanded incrementally, and the addition of processors always increases the processing power of the system. By reliability, we mean application programs continue to run, and run correctly, in spite of isolated hardware failures.

Distributed memory MIMD architectures can be applied to program solutions that generate heterogeneous processes, but can permit the use of a large number of processors. To support the goals of massive parallelism and fault tolerance to hardware failure, the Future Generation Computer Architectures project designed a distributed memory MIMD architecture consisting of approximately a million processors, each with its own local memory. The processors communicate by sending messages to each other; this is the only intercommunications mechanism provided since neither global communication nor shared physical memory is present [Harris85].

Associated with each computational processor is a special purpose routing processor. The purpose of the routing processors is to free the computational processors from message routing chores. The Ametek 2010, a commercially available distributed memory MIMD machine, uses routing processors and employs a technique called *wormhole* routing to serve the same purpose [Dally87].

An innovative interconnection topology was designed that can be used to construct robust message-passing. The topology is based on a network of orthogonal buses, and is designed with wafer-scale fabrication in mind. This topology provides so many redundant communication paths that even a relatively simple message routing algorithm can successfully route messages in a physical system where a large number of the processors have failed.

For the model of computation, object oriented programming was selected. This model has been popular in the computer simulation and artificial intelligence communities for some time. Lately its popularity has begun to spread to the general software engineering community because of the modularity and code reusability it provides for software development. In this model, the data and control functions are encapsulated in self-contained program units called objects. These objects communicate with each other by sending messages. An object can be likened to a real-world entity. It has a state that it maintains, represented by *instance variables* or memory locations. It responds to stimuli, represented by the messages it receives. It exhibits behavior in response to the messages it receives, represented by *methods* which are simply sets of program instructions. The object oriented programmer must identify the appropriate objects to represent the elements of the system and the appropriate tasks each object must perform to carry out work of the system.

The interface between the physical machine and the programmer's model of computation is the operating system. Its job is to get application programs to run on the physical machine. The operating system must provide for method execution associated with each message accepted by an object. It must provide for dynamic allocation and reclamation of objects, that is, resource management. It must provide for message routing between objects located on different processors, that is, object communication. In our case, it must do this in a manner that is tolerant of hardware failures and that does not require centralized control or access to global information.

In FY88, the Distributed Object Oriented Programming project developed a model of execution for object oriented programs that allows concurrent execution of methods not requiring serialization, while enforcing synchronization of methods that do. We can thus exploit the parallelism in object oriented programs in ways that are transparent to the programmer. In addition, this model seems capable of being extended to support fault tolerant execution of application programs.

We expect that objects will be dynamically allocated and deallocated as a program executes. We can think of objects as representing units of work assigned to processors. Memory management must provide a means of finding processors with sufficient free memory to store newly created objects. Processor management must provide a means of dynamically balancing the load on the processors in a relatively equitable manner. Since each object a processor is responsible for (at least potentially) represents a unit of work, one processor should not have to be responsible for a large number of objects while another is unused.

Consequently, we developed and simulated a resource management algorithm that meets these requirements. An important feature of the scheme is that the objects are distributed among the processors in the network in a relatively equitable manner at the time of their creation, and later redistributed when one area of the processor fabric becomes too densely populated relative to another.

As part of our resource management scheme, we devised a means to reuse the physical memory that had been freed as a result of object deallocation. The problem of deciding that an object is no longer needed by a computation is the role of the garbage collector. In the Future Generation Computer Architectures project, a distributed garbage collection scheme was developed and simulated. In FY88, the Distributed Fault Tolerant Storage Reclamation project developed and simulated a more efficient scheme.

Object relocation can be used to provide load balancing, to minimize communications distance, or to improve tolerance of hardware failure. However, the objects that need to send messages to relocated objects must still be able to communicate with them. In the course of our investigations in the Future Generation Computer Architecture project, we developed two schemes to manage object communication.

1.3 ORGANIZATION OF THE PAPER

In section 2, we describe the concurrent model of execution. In section 3, we describe the computation manager, which implements this model. In section 4, we describe the compiler, which translates a subset of Common Lisp with the Flavors object oriented extension to code executable by the computation manager. In section 5, we describe the fault tolerant resource management algorithms and the simulation of the multiprocessor system that embodies these algorithms. In section 6, we indicate our plans for future work.

SECTION 2

MODEL OF EXECUTION

2.1 INTRODUCTION

In order to develop an implementation of object oriented programming that exploits the distributed nature of a massively parallel processing environment, we must determine an internal representation for the three basic system objects: classes, instances, and contexts.

A *class* is a description of a set of application objects; it consists of the variables that represent the state and the messages to which this type of object responds. Associated with each message is a method, a set of program instructions, that represents the behavior of this type of object in response to a particular message. An instance of a class represents a particular application element. In an instance, the state variables have values that represent the state of an object at a particular time in a computation. If an appropriate message is sent to an instance, it will respond by executing the associated method encoded in the class definition of the instance. An instance is an object that maintains the state of a particular application element.

A *compiled method* is the executable version of the source code method. A *context* is an object that represents the execution state of a compiled method. It records such things as: whether the method is suspended or runnable; the next instruction to execute; and the environment the method is executing in, which includes the value of parameters passed to the method at invocation and the value of the local variables of the method as execution proceeds.

Opportunities for concurrency can be discovered in several ways. The programmer can use language constructs to explicitly identify program elements that can execute concurrently, e.g., Occam PAR and ALT statements [Perrott87], or Ada tasks [Buhr84]. The programmer can rely on a compiler to identify additional opportunities for concurrency, e.g., DO loops [Perrott87], or vectorizing [Hwang84]. However, there may be more opportunities for concurrency that can only be discovered at execution time because they depend on the data itself.

In the following discussion, we describe a model of execution for object-oriented programming that attempts to identify concurrency that can only be discovered during the actual execution of the program. We do not describe any constructs that may be provided to the programmer to explicitly generate concurrency. We do describe a mechanism that is completely transparent to the programmer as one means of generating a great deal of concurrency. We also describe a mechanism for managing concurrency, regardless of how it is generated.

2.2 A CONCURRENT MODEL OF EXECUTION

The runtime behavior of a program with a given input data set can be represented as a directed graph with cycles (figure 2-1). The circles in this figure represent objects, and the arrows represent messages. When an object receives a message, the method associated with that kind of message begins executing. If we execute this program on a sequential processor, the labels on the arrows represent the order in which the messages are processed. Every arrow represents a request for processing. Associated with each request is a reply message that is not shown.

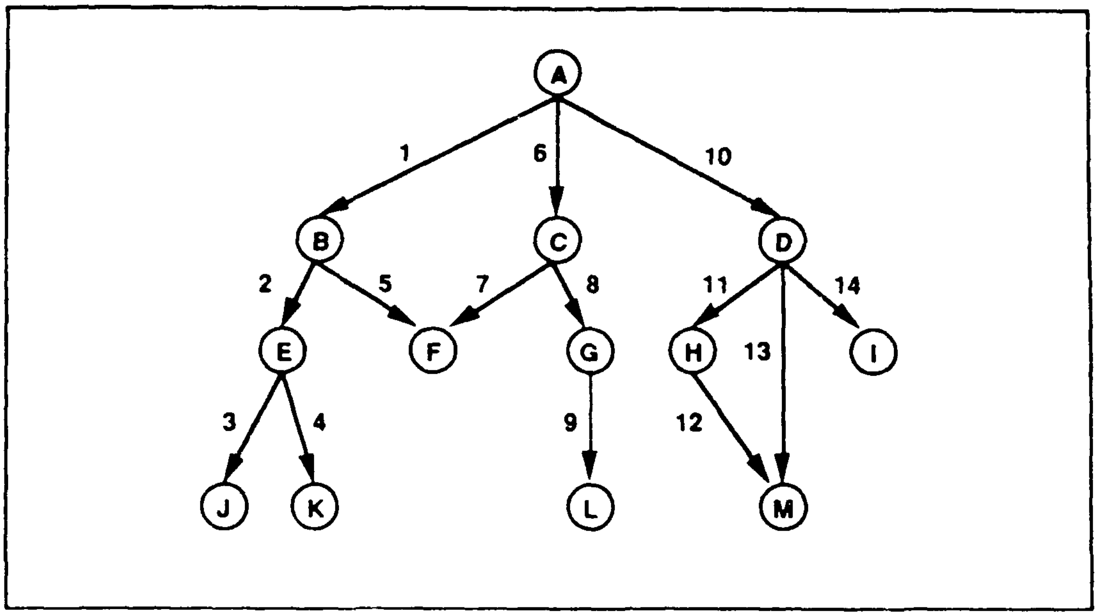


Figure 2-1. Serial Execution of the Program

We may attempt to make this program execute faster by running it on a multiprocessor. If objects do not reside in the same processor, it may be possible for them to process messages concurrently. However, we must ensure that the parallel execution yields the same result as the original serial execution. This requirement places constraints on the way multiple messages are handled by a single object.

Suppose we have an object A with state variables x, y, and z, and the method in figure 2-2. Suppose the value of x is an object B, and the value of y is an object C. Then line 1 says "Send the message :m1 to B, and assign to x-result the value that B returns." Similarly, line 2 says "Send the message :m2 to C, and assign to y-result the value that C returns."

```
(1) x-result := send x :m1

(2) y-result := send y :m2

(3) z := x-result + y-result
```

Figure 2-2. A Method

Since we do not need the results of these messages until line 3, it may be safe to send both messages and have them processed concurrently. At line 3 we have to wait for both results to be returned before the method can proceed.

```
(1) x-result := send x :m1

(2) y-result := send y :m2 (x-result)

(3) z := y-result
```

Figure 2-3. Another Method

Now consider another method (figure 2-3). In this method the value of x-result is passed as an argument in the :m2 message to C. It appears the :m2 message cannot be sent to C until the reply from the :m1 message to B has been received. However, we can continue processing by using a special kind of object called a *future object* that plays a role similar to Futures in Actors [Agha86], MultiLisp [Halstead85] or MultiScheme [Miller87] A future object acts as a place holder for the result of a computation. It has one state variable and two messages, :get-value and :set-value. In the method of figure 2-3, A can still send both messages and have them processed concurrently. Before it sends the first message, however, it creates a future object F and sets the value of x-result to F. When it sends the :m1 message to B, it passes a pointer to F in a special field of that message. Instead of returning its result to A, B sends it to F in a :set-value message.

For some operations, C may be able to use the future without knowing its actual value. For example, C may pass F as an argument in a message to another object, or it may assign F as the value of one of its state variables.

If C needs the actual value of B's result to perform an operation, e.g., addition, then C must be able to recognize it is holding a future. At that point, C sends a :get-value message to F. If F has not received a :set-value message before it receives the :get-value message, F postpones responding until it has received and processed the :set-value message.

2.3 A CORRECT MODEL OF EXECUTION

Let's return to figure 2-2. We have already observed that it appears to be safe to send the messages in lines 1 and 2 to be processed concurrently. But in fact this may not be true. If x and y refer to different objects, say B and C, that do not send any messages when they process their respective :m1 and :m2 messages, it is safe.

On the other hand, if B and C send messages to another object (figure 2-4), it might not be safe. Suppose B computes for a long time before sending the :m3 message, while C only computes for a short time before sending the :m4 message. The :m4 message may arrive and be processed before the :m3 message arrives. It might still be safe as long as neither message writes a state variable that is read by the other. Otherwise, it may be important that the :m3 message be completely processed first.

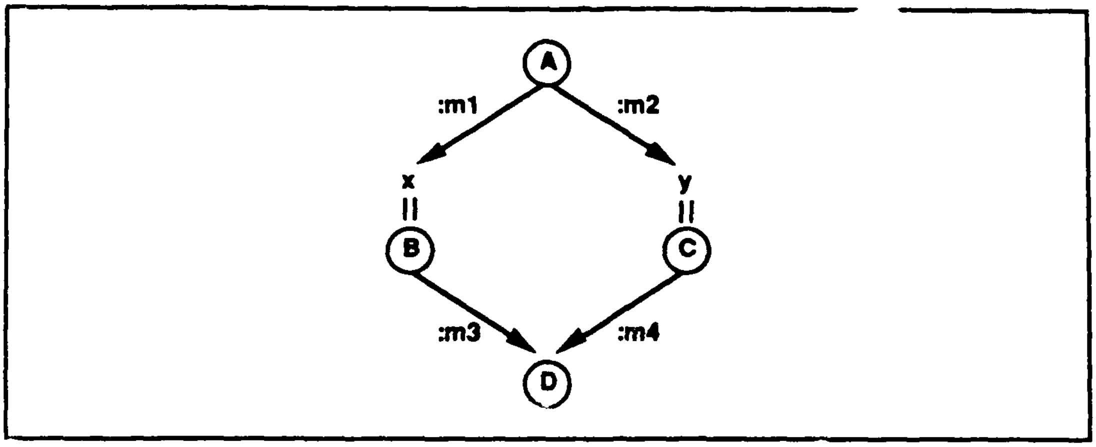


Figure 2-4. Different Objects Refer to the Same Ooject

With recursion, it is not possible to finish processing one message before beginning another. In figure 2-5, object A receives an :m1 message. While processing that message it sends an :m2 message to B. While processing the :m2 message, B sends an :m3 message to A. A cannot complete its :m1 processing until B completes its :m2 processing, and B cannot do that until A processes the :m3 message. A must process the :m3 message with its state variables having the same values as when it sent the :m2 message to B. Furthermore, any computation A performs in its :m1 method after it sends the :m2 message must begin with its state variables having the same values as when the :m3 method completed.

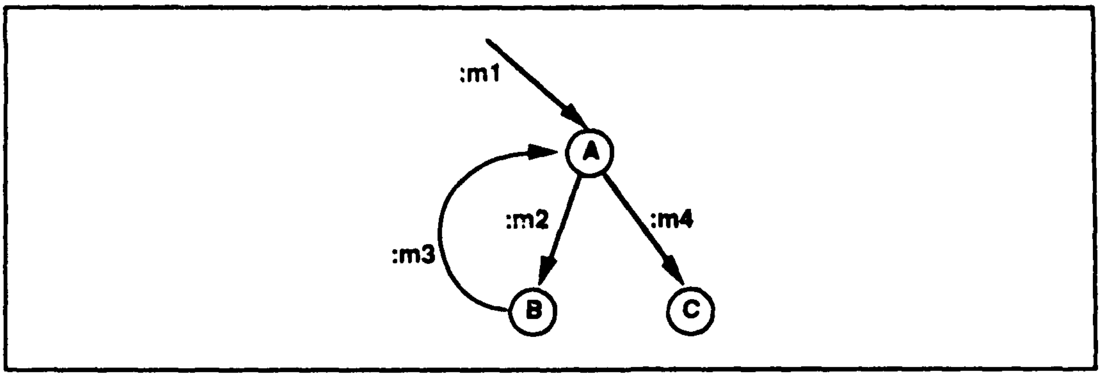
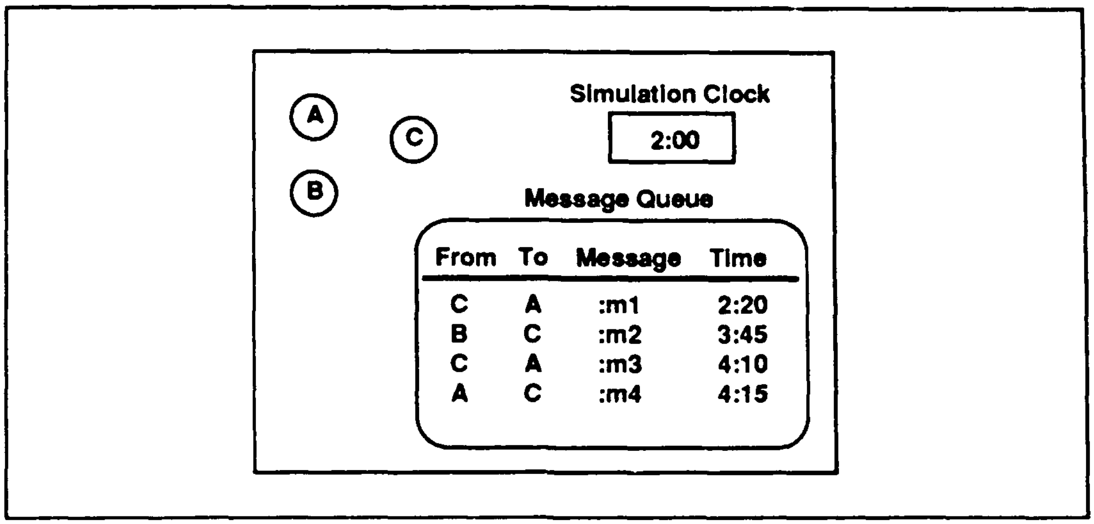


Figure 2-5. Recursion

2.3.1 Ideas from Distributed Simulation

To satisfy the constraints on how multiple messages are handled by a single object,we borrow an idea from the world of distributed discrete event simulation. In object-oriented discrete event simulation on a single processor, there is a set of objects that represent the real-world entitie, being simulated, and a single message or event queue that is kept in simulation time order (figure 2-6). An entry in the message queue indicates what event is to occur, at what time, and to which object. A message is taken from the head of the queue, the simulation clock is moved up to the time of the event, and the appropriate method is executed. The execution of this method may cause its object's state to change, add one or more messages to the message queue, or delete one or more messages from the message queue.



From	To	Message	Time
C	A	:m1	2:20
B	C	:m2	3:45
C	A	:m3	4:10
A	C	:m4	4:15

Figure 2-6. Discrete Event Simulation Single Processor

Essentially two strategies have been proposed for synchronizing the execution of these simulations on multiprocessors. In both strategies, objects and the associated parts of the message queue are distributed among a number of processors (figure 2-7). Speedup comes from allowing more than one object to process a message at a time. With a single simulation clock, only objects with messages at the current simulation time can execute concurrently. It is assumed that more speedup comes from giving each object its own simulation clock, and allowing different objects to concurrently process messages at different simulation times.

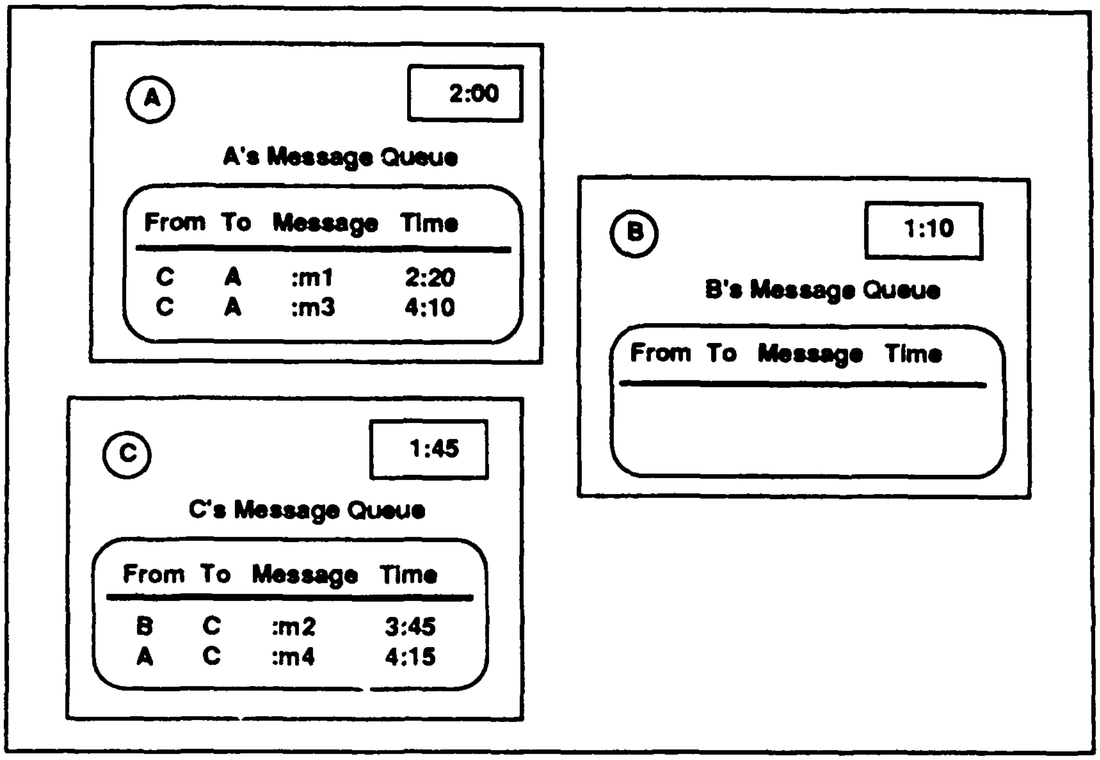


Figure 2-7. Discrete Event Simulation Multi-Processor

Both strategies distribute the simulation clock and allow each object to keep track of its own simulation time. In figure 2-7, for example, when A accepts the :m1 message, it updates its clock to 2:20. If, at the same time, C accepts the :m2 message, it updates its clock to 3:45. However, if A's :m1 method causes A to send a message to be processed by C at 3 o'clock, and that message changes the way C's 3:45 message is processed, then C has made a mistake in advancing to 3:45.

Misra, Chandy and others [Misra86] have suggested schemes that force C to block or suspend until it is safe to go ahead. In order to avoid deadlock, or to resolve it when it occurs, these schemes require that messages sent from the same source object be received in order, and that potential relationships among the objects in the simulation be known in advance. C cannot advance to 3:45 if some object may send it a 3 o'clock message.

The problem with this method is that, in order to avoid deadlock or to resolve deadlock when it occurs, potential relationships among all objects must be known in advance. It may be possible to know all relationships in a simulation, but in general-purpose computation, where objects may be created dynamically, it may not.

Jefferson has suggested an optimistic technique called Time Warp [Jefferson82, 85, 87]. In Time Warp, C processes messages as they arrive, but before processing a message it saves its state. For example, when the 3:45 :m2 message arrives, C saves its 1:45 state and processes that message. Now if a 3 o'clock message arrives, C rolls back to its 1:45 state, advances its clock to 3 o'clock, processes the 3 o'clock message, saves a new 3 o'clock state, and then reprocesses the 3:45 message.

The basic rollback mechanism uses three structures for every object: an input message queue, an output message queue, and a state stack. The input queue contains messages sent to the object. It is ordered by message *receive* timestamp (the time at which the message is to be processed). The queue contains messages that have already been processed as well as those that have yet to be — the object's clock identifies where it is in the input queue. When a message arrives with a timestamp earlier than the object's current time, the object has to roll back to a time no later than the time of the new message, process that message, and then reprocess all the intervening messages. The output queue contains copies of all messages sent by the object. It is ordered by message *send* timestamp (the value of the object's clock when it sent the message). When the object rolls back, it has to be able to undo state changes it may have erroneously induced in other objects. It does that by sending what are called *anti-messages* to cancel any output messages it has to roll back over. The state stack records the object's state — its time, and the values of its state variables at that time — before each input message that was processed. When the object has to roll back, the appropriate state is restored from this stack.

With Time Warp, messages need not be received in correct order, and potential relationships among objects need not be known in advance. These features make it appealing as a means of handling synchronization in distributed general-purpose computation. In a message-passing MIMD, like the Intel iPSC, there may be overhead in ensuring that messages from the same source are delivered in the order sent. But more importantly, in general-purpose computation we expect relationships among objects to change frequently, new objects to be created dynamically, and even classes of objects to be created or modified dynamically.

Jefferson suggests a mechanism to reduce the amount of old state information that need be retained [Jefferson85]. Essentially, the computation is always moving forward; *i.e.*, there is a simulation time, called the *global virtual time (GVT)*, past which the computation can never roll back. According to Jefferson [Jefferson85],

> "It can be proved that the theoretical definition of GVT for an instantaneous snapshot can be characterized operationally as the minimum of (a) all virtual times in all local virtual clocks in the snapshot, (b) all virtual send times in unreceived messages in the input queues of the snapshot, and (c) all virtual send times in messages that have been sent but not yet acknowledged (and may, therefore,be in transit at the moment of the snapshot)."

State information with timestamps earlier than GVT can be discarded. For example, if an object has states with timestamps 1, 2, 3, 4, and 5 o'clock and GVT is 3:30, then the states with timestamps 1 and 2 o'clock and messages with send times before 3:30 may be discarded.

GVT is also essential to knowing when to commit to exception handling and I/O. When the execution of a method causes a runtime error to occur, that error should not abort the program as a whole. The object that executed the method may yet receive a message with an earlier timestamp, be rolled back to the earlier time, and not encounter the error condition again. The runtime system can only commit to an error when it occurred at a time earlier than GVT.

Similarly, the runtime system cannot commit immediately to external program output. The user should only be able to see the system at GVT — the only time at which it is guaranteed to be in a consistent state. The program should only be able to affect the external world when it is fully committed to its actions.

It is perhaps less intuitive that the user herself should not be allowed to inject messages into the system at arbitrary times. If the user is permitted to send a message at a time earlier than GVT, an object may have to roll back to a time earlier than the oldest state that is being saved. Thus, for an interactive simulation, the user must be included in the decision process that determines when all objects agree that GVT can be advanced.

2.3.2 Computation Time

In simulation, the time at which actual events occur, relative to each other, is used to manage the ordering of simulation events. In general-purpose computation, we need a substitute for time that can be used to synchronize the computation. A set that is well ordered under the relation of lexicographical ordering will serve our needs, for example, the set of character strings. Suppose an object W receives a message with the timestamp "accb." Then every message W sends while processing that message has a timestamp prefixed by "accb" but with at least one additional character appended, e.g., "accba". For each message W sends, the character code of the appended character is increased to indicate that, in terms of the computation, this event occurred after the former event. For example, if W sends one message with timestamp "accba", the next message W sends has timestamp "accbb." These timestamps form a well-ordered set that can be used to order multiple messages to individual objects to agree with a serial execution of the same program (compare figures 2-1 and 2-8).

Notice in figure 2-8 that F can decide which message (the one from B or the one from C) to process first, since the timestamp on the message from B is "ab," the timestamp on the message from C is "ba", and "ab" is less than "ba". Suppose F receives the "ba" message first and mistakenly processes it. When it receives the "ab" message it rolls back, processes the "ab" message, saves a state, and then reprocesses the "ba" message. Similarly, M can decide which message to process first because "caa" is less than "cb".

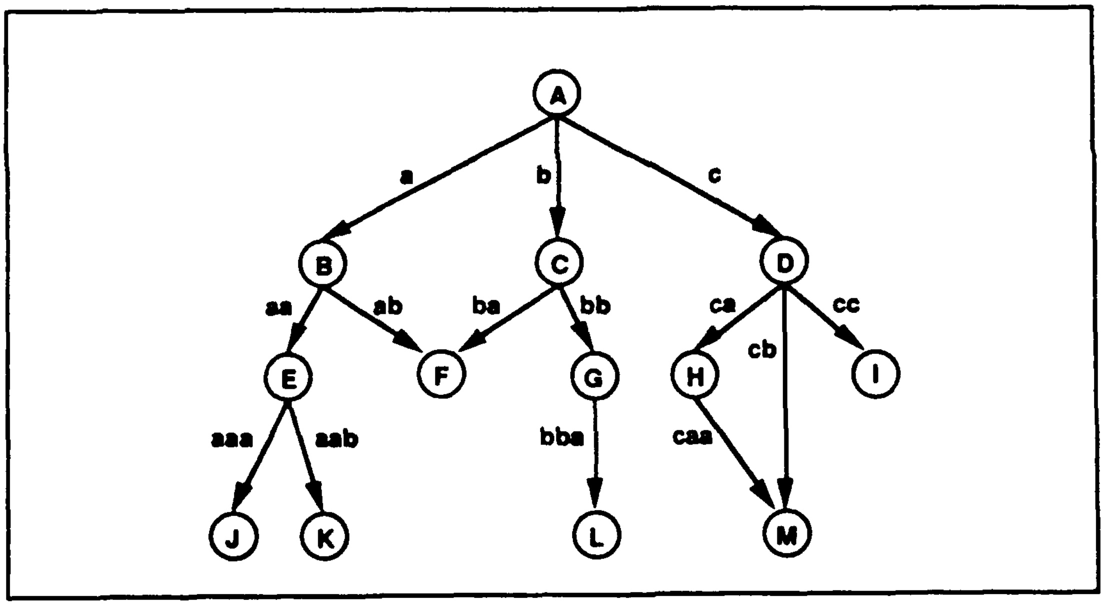


Figure 2-8. Timestamps

2.3.3 Handling Replies and Recursion

In Time Warp [Jefferson87], there are two kinds of messages that can be sent — *query* messages and *event* messages. A query message is simply a request for information. When an object receives a query message, the method it executes may access, but not alter, the state variables of the object and may send query, but not event messages, to other objects. When an object receives an event message, the method it executes may access (read) and alter (write) the state variables of the object, and may send query or event messages to other objects. Thus, event messages may have side effects, query messages may not.

Every query message returns a *reply* message to the object that sent the query. When a method sends a query message, it suspends until the reply is received. Event messages do not return replies. When a method sends an event message, it continues executing. Thus, the result of an event message cannot be used later in the same method.

With every query message a method sends, the system automatically associates a receive timestamp. A query's receive timestamp is equal to the current simulation time of the sending object, the simulation time *now*. For every event message a method sends, the programmer must specify a function to calculate a receive timestamp (the time at which a message is to be processed).

Query messages are always processed before event messages with the same receive timestamp. That is, the reply to a query message with receive timestamp *T* reflects the state of the object before any event messages with receive timestamp *T* have been processed.

In Time Warp, a cycle of recursive query messages all with the same receive timestamps is allowed. However, a cycle of recursive event messages all with the same receive timestamps is prohibited. The purpose of this restriction is to avoid Time Warp's equivalent of deadlock — infinite rollback.

These restrictions may seem natural in the context of simulations of real-world situations; however, they force a programmer to structure general-purpose programs in an unnatural way. Consider the method associated with the message :m0 in figure 2-9. Suppose we wish to translate this simple method into the Time Warp programming model. If the processing of the :m1 message has no side effects, then the Time Warp translation in figure 2-10 does the job. The system will automatically associate the receive timestamp *now* (the current simulation time of the sending object) with the query message. After the method in figure 2-10 sends the query message :m1, it suspends processing until the result of the query is returned.

```
define-method for the message :m0
   R-result := send R :m1
   y := R-result + x
end.
```

Figure 2-9. A Simple Method

```
define-method for the message :m0
   R-result := query R :m1 now
   y := R-result + x
end.
```

Figure 2-10. Time Warp Translation for a Query

Suppose the :m1 message does cause side effects. Suppose it alters one of the receiving object's instance variables and must report back on the new value. Since an event message cannot return a result, the programmer might consider sending a query message (following the :m1 message) to the object R to retrieve the result. But the system will associate a receive timestamp of *now* with this query; thus, the reply to this query will reflect the state of the object R before the :m1 event message had been processed.

Another approach is for the programmer to break up the original method into two methods, (figure 2-11), and alter the receiving object's method. In the first method, the :m1 event message is sent. When the receiving object completes the computation associated with :m1, it sends an event message :finish-m0, with the required result as an argument and with timestamp *later*, to the object that sent it the :m1 message. When the object receives the message :finish-m0, it performs the rest of the original :m0 method.

```
define-method for the message :m0
   event R :m1 now
end.

define-method for the message :finish-m0 (R-result)
   y := R-result + x
end.
```

Figure 2-11. Time Warp Translation for an Event

The problem with this approach is to decide how much later *later* should be than *now*. The programmer might try to make *later* equal to *now*. In Time Warp, a cycle of query messages all with the same receive timestamps is allowed. However, a cycle of event messages all with the same receive timestamps is prohibited. So *later* must be later than *now*.

The programmer must be very careful in the selection of *later* to ensure that another event message with a timestamp between *now* and *later* is not processed by the object (figure 2-12). An intervening event message might change the state of the object so that the :finish-m0 method is not processed correctly. Thus, although it is possible for a programmer to work around the restriction on event messages not sending replies, it is by no means trivial.

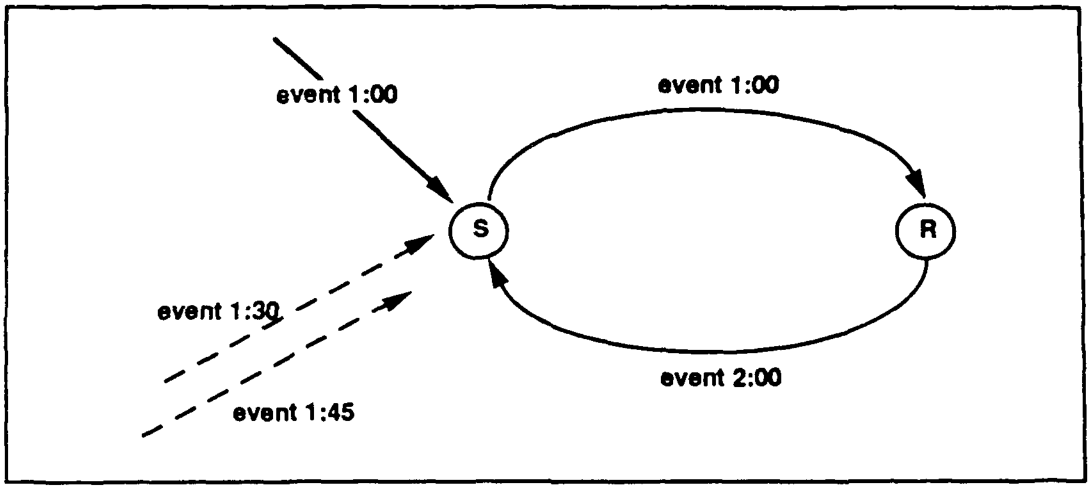


Figure 2-12. Time Warp Event Replies

In Time Warp, a cycle of recursive query messages all with the same receive timestamps is allowed. However, a cycle of recursive event messages all with the same receive timestamps is prohibited. This restriction makes side-effecting recursion — which is a useful programming technique — difficult to do. It requires the programmer to manage the timing of events so that no intervening messages are processed while the recursion is in progress (figure 2-13). This is not an easy task as it may be hard to predict (until execution time) the depth of a recursion and, hence, the number of messages involved.

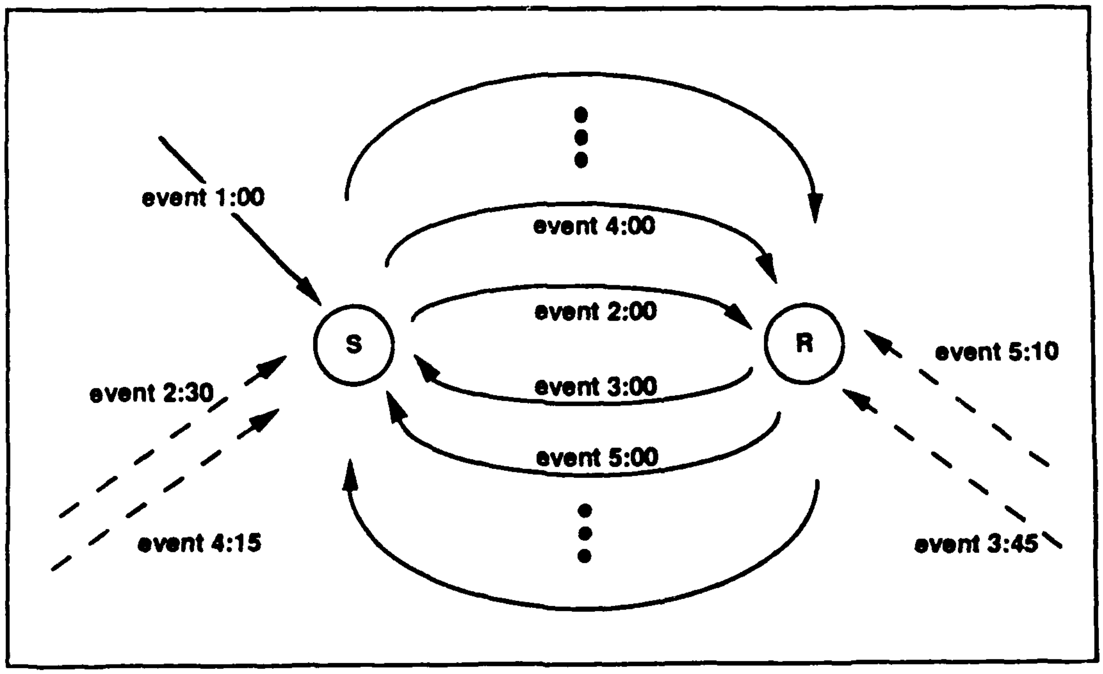


Figure 2-13. Time Warp Event Recursion

In our model, a sending object can use the result of a side-effecting message it sent later in the same method, and side-effecting recursion is fully supported. There are two reasons for this. First, our model of execution controls time while the program is executing (not the programmer before the program is run). Our computation time dynamically and automatically attains as fine a granularity as necessary to support replies from side-effecting messages and side-effecting recursion (figure 2-12). Second, our model allows method execution to be rolled back so that side-effecting recursion can be handled correctly.

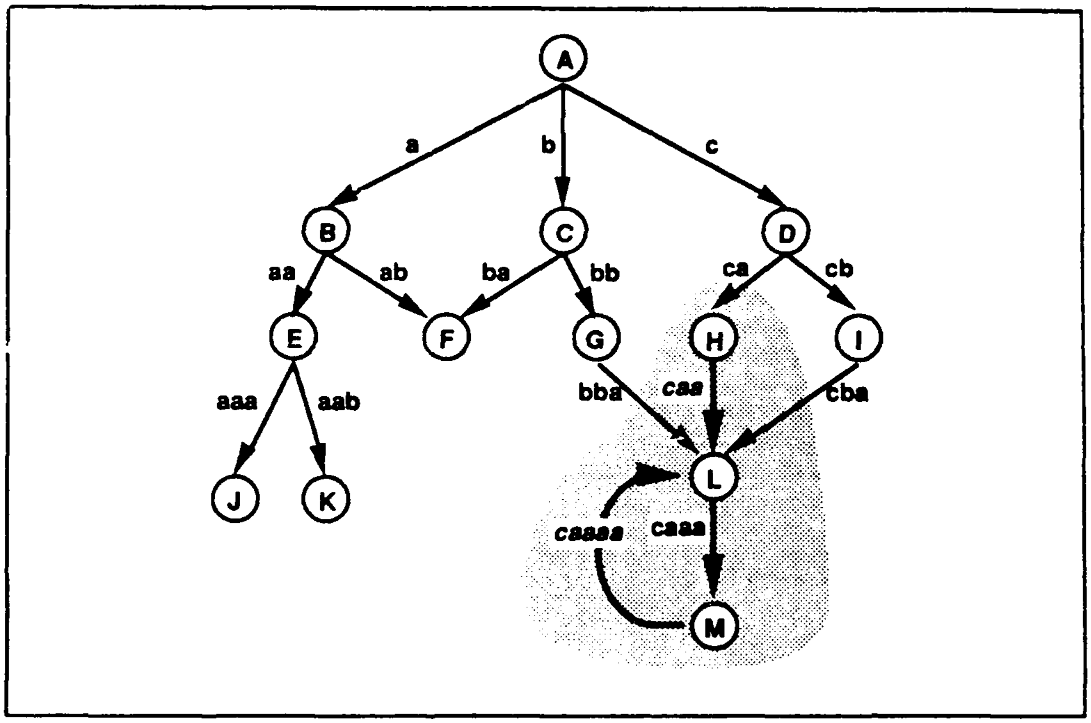


Figure 2-14. Timestamps with Recursion

In figure 2-14, we see that there are four objects that send messages to the object L: G, H, I and M. Suppose that the processing of the message from H to L results in L sending a message to M, and M's processing of that message causes M to send a message to L, that is, recursion occurs. However, let us also assume that none of the other messages to L causes a recursion to occur.

L has to be able to identify three different situations: a message with an earlier timestamp than the one it just processed or is currently processing; a message that indicates a recursion is about to take place; and a message with a later timestamp that is not a recursion. If the timestamp of the message L is processing is greater than the timestamp of the incoming message, then L must rollback and process the earlier message.

If the timestamp of the message L is processing is less than the timestamp of the incoming message, then L must decide if the incoming message should be processed after the current message has been completely processed or if recursion is taking place. When a recursion occurs, the processing of the method associated with the current message is suspended at a point just after it sent the message that initiated the recursion. The recursive message begins processing with the instance variables in the state they were in just after the message that initiated the recursion was sent. The recursive message must be processed completely before the current message can continue processing. When the current message resumes processing, the state of the instance variables must be as they were at the completion of the recursive message.

In the example, suppose L is processing the "caa" message from H. If L receives the "bba" message from G, L rollbacks. If it receives the "cba" message from I, L keeps the message queued until it has completely processed the "caa" message. If L receives the "caaaa" message from M, it recognizes that a recursion is occurring because the timestamp of the current message "caa" is a prefix of the timestamp of the incoming message "caaaa."

SECTION 3

COMPUTATION MANAGER

3.1 INTRODUCTION

Our model of execution uses both futures and rollback to synchronize elements of the computation when necessary. The details presented here correspond reasonably well to our initial implementation in a simulator used to test and experiment with the design as it develops.

To aid us in our understanding of the problems involved, we have developed code for a computation manager based on our model of execution that runs on a simulation of a multiprocessor. To ensure that the model of execution will have wide applicability, the simulation makes very few assumptions about the multiprocessor system, except that message delivery is reliable, that is, messages are always delivered correctly, but not necessarily in the order sent.

3.2 TIME MANAGED OBJECTS

An application object is an object defined in the programmer's code. A context object is an object associated with a particular invocation of an application method. An application object or context object that changes state as the computation proceeds, must be *time managed* by the computation manager. (Read-only application objects do not require time management.) Every application or context object that requires time management, is wrapped in a *Time Warp* object (figure 3-1). Each Time Warp object has three instance variables: current-state, input-queue, and old-states stack. The *current-state* and each element in the old-states stack is a *State* object with instance variables: time, the application or context instance variables, the message-processed at this time (message receive-time equals the state's time), a list of the messages-sent at this time (message send-time equals the state's time).

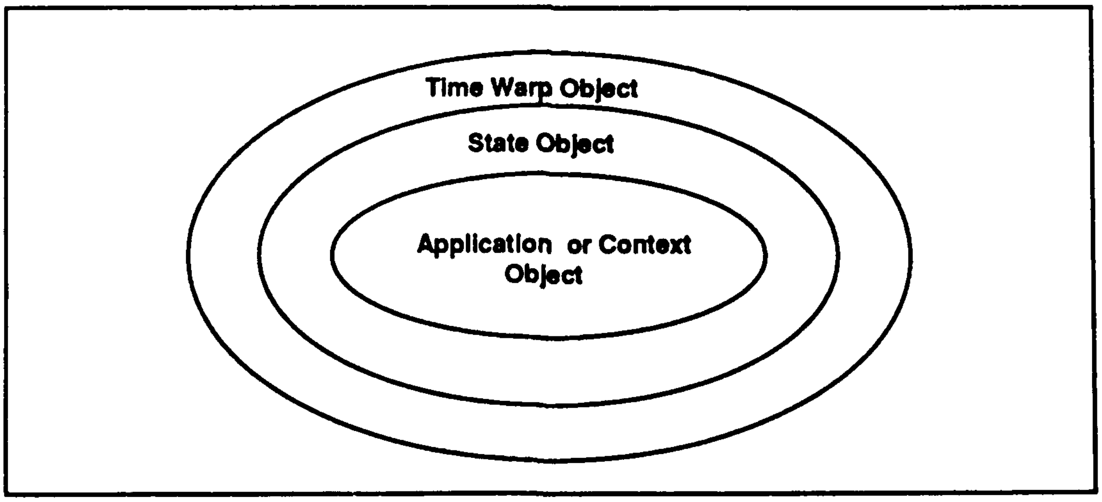


Figure 3-1. Time Managed Objects

An application object has instance variables that include: the instance variables specified in the application programmer's code, and a context stack that is used to manage the contexts associated with messages currently being processed by the object. A context object has instance variables that include:

the name of its associated Time Warp application object, the instance variables specified in the application programmer's code, the arguments associated with its method, a pointer to the array that embodies the compiled code of the method, an instruction pointer into that array that indicates the next instruction to be executed, variables for managing local environments established within a method, the timestamp of the request message that initiated the context, the timestamp suffix for the next request message to be sent by the context, and a variable that indicates whether the context is waiting for the value of a future object.

Each time an object receives a request message, it creates a context object (wrapped in a Time Warp object as described above) to manage the execution of the application method associated with the message.

3.3 DYNAMICS OF METHOD EXECUTION

As a result of method execution, one object may send a message to another object to request processing. When an object receives a request message, it saves its state and creates a context to manage the execution of the appropriate method. After creating a context, the object sends it a :start message that contains the timestamp and argument values in the request message, and copies of the object's state variables. When the context receives this :start message, it begins executing its method. When the method completes executing, the context sends the object a :done message that contains new values for the object's state variables. Until it receives this :done message, the object does not normally accept additional messages with later timestamps: the first message must be processed to completion before the next message is taken from the input queue. An exception to this rule is in the case of recursion, as we shall explain later.

Figure 3-2 illustrates the dynamics of method execution. In this figure, object A receives a request message with timestamp "j" (Q-j). It creates context A-C-0 and sends it a :start message with timestamp "j" (S-j). While executing its method, A-C-0 sends request messages to objects B and C. By the way we determine computation time, these messages are sent with timestamps "ja" and "jb." B and C handle their messages similarly to the way A handles its Q-j message. When A-C-0 has completed executing its method, it sends a :done message (D-k) to A with new state variable values. The timestamp on the :done message is the successor to the timestamp on the :start message: A-C-0 begins executing its method at time "j" and finishes at time "k." When A receives the :done message, it updates its state variables and takes the next message from its input queue.

Before a method sends a request message to another object, the computation manager creates a future object to hold the result of the computation performed by the processing of the message. Figure 3-2 also shows a future object, Fut-1, that is used to hold the result of B's computation. Before A-C-0 sends the request message (Q-ja) to B, the computation manager created the future object Fut-1. When A-C-0 sends the message to B it passes a pointer to Fut-1 in a special field of the message. When A-C-0 sends the request message (Q-jb) to C, it passes a pointer to Fut-1 as an argument. After processing A-C-0's request, B-C-1 sends its result to Fut-1 in a :set message (set-jaa). If C-C-2 ever needs the actual value of its argument, it sends a :get message (get-jba) to Fut-1. If Fut-1 has already received the :set message from B-C-1 when it receives the :get message from C-C-2, its value is returned immediately to C-C-2 (R-jbb). If Fut-1 has not received the :set message from B-C-1, it postpones sending a reply to C-C-2 until its value has been set.

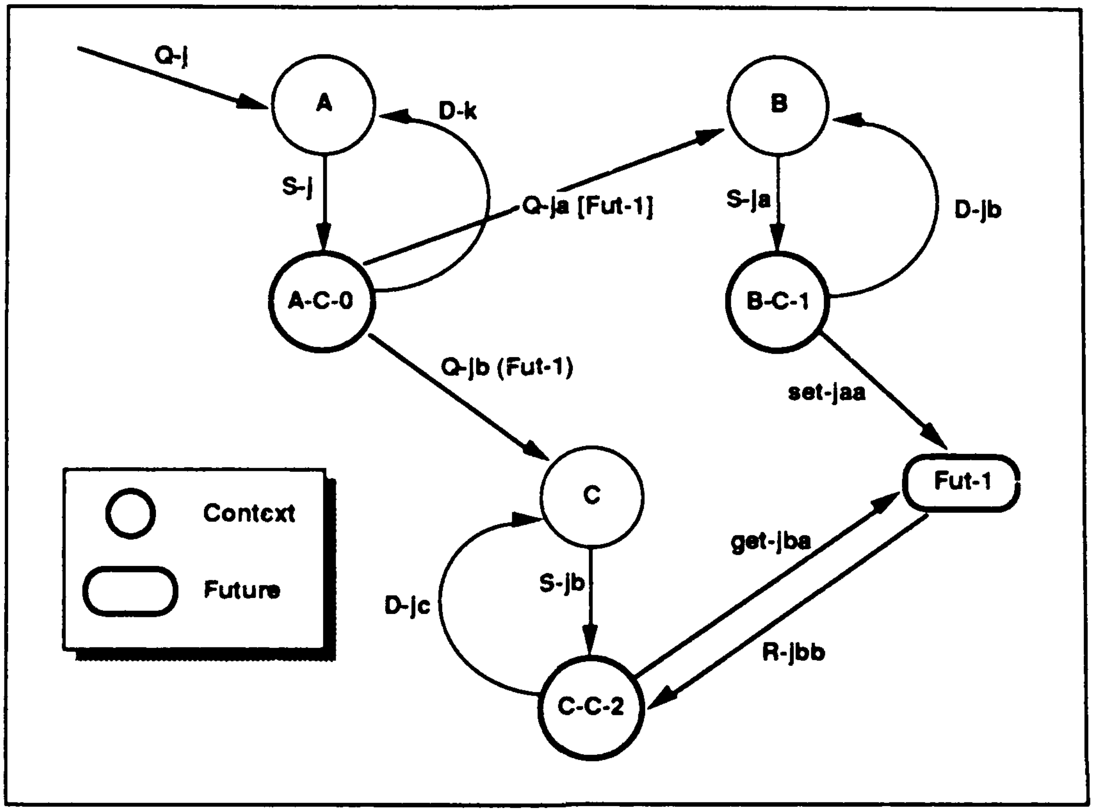


Figure 3-2. Method Execution

3.4 RECURSION

The mechanism described thus far is not capable of handling recursion, which requires an object be able to suspend the execution of one method, receive another message, and execute to completion the method that corresponds to the new message, before it continues the first method. Recursion occurs if A is sent a request message by A-C-0, or by some other context as a result, direct or indirect, of a request message sent by A-C-0. Figure 3-3 illustrates how recursive method execution is performed. A-C-0 sends a request message with timestamp "ja" to A. Ordinarily, request messages with timestamps greater than the current message remain in A's input queue until the method has completed execution and A's state variables have been updated. But in this case, A can tell it is in a recursion and should take the Q-ja message, because its current message timestamp, "j," is a prefix of the new message's timestamp.

With recursion, an object must have more than one context active at the same time. To handle this, every object maintains a stack of contexts as part of its state. When recursion occurs, the stack contains a context for each level of recursion, as well as one for the original request message.

In figure 3-3, when A receives the recursive Q-ja message, it creates context A-C-1 and pushes the new context onto its context stack. Then A sends the context that was previously on the top of the stack, A-C-0, a :send-values message with the timestamp "ja" (SV-ja) and a pointer to A-C-1.

When A-C-0 receives the SV-ja message, it rolls back to its state immediately after sending the Q-ja message. It then sends A-C-1 a :start message with the values that its copies of A's state variables had at that time. This ensures that the method A-C-1 sees the correct initial values for A's state variables. When A-C-1 finishes executing its method, and sends A a :done message with new values for its state variables, A forwards those values to A-C-0 in an :update-values message (UV-jb). After rolling back to its "ja" state, A-C-0 blocks until it receives the UV-jb message from A. Then it resumes its method with its copies of A's state variables correctly reflecting the complete processing of the recursive message.

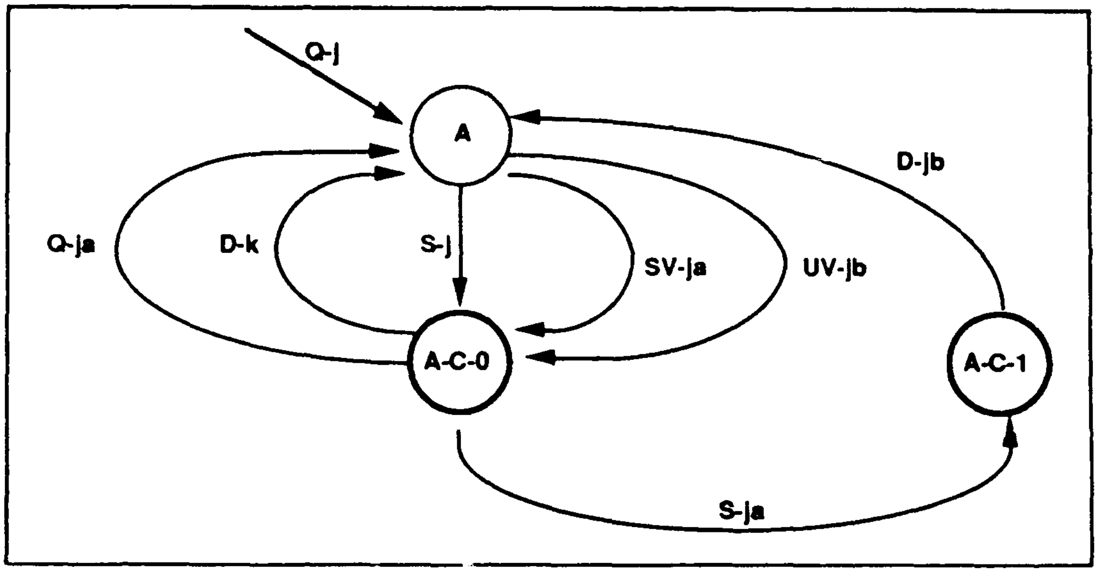


Figure 3-3. Method Execution With Recursion

3.5 CURRENT STATUS OF THE COMPUTATION MANAGER

The computation manager was written in Common Lisp with the Flavors object oriented extension. In addition to the assumptions mentioned previously with regard to the multiprocessor system, it assumes that every processor in the multiprocessor machine has a Common Lisp runtime system that supports the Flavors object oriented extension. This is not an unrealistic assumption given that both the Ametek 2010 and the INTEL iPSC/2 plan to implement Common Lisp on their systems by providing each node with a Common Lisp runtime system, and that Flavors is simply an extension to Common Lisp given by macro definitions. The computation manager would run on top of this runtime system.

We assume the application programmer is using a subset of Common Lisp with the Flavors object oriented extension as the application language. Classes are defined by the application programmer as Flavor definitions with associated methods written using Flavors method defining macros.

One of our goals has been to investigate concurrency that the system can provide in a manner that is completely transparent to the application programmer. At the present time no mechanisms in the form of extensions to the application language are provided. For example, futures are generated and managed exclusively by the operating system. It is not that we do not envision such extensions. It would probably be useful to enhance the potential concurrency of a computation to have a statement that would allow the application programmer to specify that a set of messages may be processed *truly* simultaneously (that is, with the same receive timestamp). It might be useful to allow the programmer to ask questions about the state of future values; for example, an extension that allowed a computation to proceed as soon as one variable of a set of variables is not a future value. However, at the present time we are simply not attempting to support such extensions. In part at least, because the program might then be subject to concurrency generated errors that the computation manager is currently protecting against.

A compiled method is the executable version of the source code method. Compiled methods are arrays of program instructions. Since we assume the runtime system is Common Lisp, these program instructions are Common Lisp expressions.

There is nothing inherent in the Common Lisp language that makes it impossible to use another language. It is simply that the language provides many features that are useful for rapid implementation of prototype systems, and it is readily available on many different single processor architectures and will soon be available on a number of multiprocessor systems. Currently the computation manager runs on top of a simulation of a multiprocessor architecture on both the Sun running Franz Common Lisp with Flavors and the Macintosh running Coral Common Lisp with Flavors.

3.6 COMPUTATION MANAGER OVERVIEW

Before an application program can be run, it must be complied into a form executable by the computation manager. Application methods are translated by the compiler into arrays of program instructions. These instructions include application program instructions and special directives to the computation manager. The directives are usually associated with context object methods, for example, :send-request, :send-done, :push-local-variables, :pop-local-variables.

The multiprocessor simulation uses a message bag to simulate the communication system. When an object sends a message to another object the message in added to the message bag. The simulated operating system consists of a loop. First, one or more messages (user specified) are randomly selected from the message bag and delivered to the appropriate Time Warp application or context object's input queue. A message contains several fields that include: the object to which the message is directed, the object that sent the message, the send timestamp, the receive timestamp, the message type (for example, request, start, done, send-values, update-values), and another field that contains the text of the message. Next, each Time Warp object is given an opportunity to process a message from its input queue. Finally, each Time Warp context object is given an opportunity to execute a number of instructions in its method. The number of instructions executed may be user specified. In the simulation, this number is meant to represent the amount of processing a processor can perform relative to the time it takes to send a message. A Time Warp application or context object saves its current state on its old state stack before it processes a message. In addition, if the execution of an instruction causes a Time Warp context object to send messages to other objects or to establish or disestablish a local environment, the context object will save its current state on its old state stack after it processes the instruction.

When a Time Warp context object gets an :execute-instruction message from the simulated operating system, it evaluates the element of the compiled method array that the instruction pointer is pointing to in the scope of the instance variables of the context object. This instruction may include operations that affect the instance variables of the object, arguments of the method, or currently active local variables. In addition, the instruction may include operations that cause invocation of one of the other methods associated with a context, for example, :send-request, :send-done, :push-local-variables, :pop-local-variables. When the application method code requires that a message be sent to another object, this request is handled by the invocation of the context method associated with the :send-request message. When the application method has completed executing, the context method associated with the :send-done message handles notifying the object associated with the context of the completion of the method and the current state of the object's instance variables.

The :push-local-variables and :pop-local-variables context methods handle the establishment and disestablishment of local environments within an application method. The :push-local-variables context method defines a new class of context object that contains the newly defined, appropriately initialized, local variables as instance variables. In addition, the list of names of the new local variables are stacked on a special context variable, current-locals. If a new local variable has the same name as a currently active local variable or application instance variable, the name and old value pair is appended to the old-locals (another special context variable) list. The name of the old context class name is also stacked on another special context variable, old-local-classes. The old context state is saved on the old-states stack. A new context state is created that contains a new context object that is an instance of the newly defined context class. This new context state becomes the current state.

The :pop-local-variables context method creates a new instance of the context class that was operative before the last local environment was established, that is, before the last :push-local-variables was executed. The name of this class is obtained by popping the old-local-classes stack. The names of the last set of local variables that were stacked is obtained by popping the current-locals stack. The old-locals list is searched to see if it contains any of the names in the last set of local variables. If it does, the old variable's value is popped from the old-locals stack and made the value of the variable in the new context object. The old context state is saved and the new state that contains the new context object is made the current state.

3.7 RECOGNIZING FUTURE VALUES

A future object acts as a place holder for the result of a computation. A future can be assigned as the value of a variable or passed as an argument in a message to another object. Only when the actual value of the result of the computation is needed must it be retrieved from the future object.

For certain operations, the Lisp runtime system will return an error if one or more of the operands is a future value. For example, attempting to add a future value causes the system to signal an error. Under other circumstances, this is not the case. For example, in the implementation of Common Lisp on both the Sun and the Macintosh, the IF statement merely tests if its first argument is not null.

In the current implementation of the computation manager, if the Lisp runtime system signals an error because a future value is inappropriately presented as the argument to a function, the computation manager checks all the arguments to the function (as specified by the compiler) and automatically sends messages to all future objects that are associated with future values that appear as arguments. The context is then put in a wait state until the actual values are available, and then the statement is executed again.

For functions that do not generate errors when their arguments fail to meet type requirements, the compiler generates code that allows the computation manager to test whether any of the arguments are future values before the instruction is executed (:test-futures). When the actual values of all future values have been received, the statement is then executed.

3.8 RECOGNIZING RECURSION SYNCHRONIZATION ERRORS

Because the computation is always pressing forward, values of variables may be the wrong type when a statement is executed. Let us consider the following example. Suppose A is an instance of my-class with instance variable iv1, and B is another object, and A has the method given in figure 3-4.

```
                (defmethod (my-class :help) ( )
        (1)         (setq iv1 1)
        (2)         (send self :set-iv1 B)
        (3)         (send iv1 :hello)
                    )
```

Figure 3-4. Recursion Synchronization Errors

The first statement sets iv1 to the value 1. In the second statement, the variable *self* contains a pointer to the instance A itself. In Flavors, this variable is used to allow an instance to send a message to itself to invoke the same or a different method. Thus, the second statement causes a recursion to occur by sending a message to A that sets its instance variable iv1 to an object B. Finally, the last statement sends a :hello message to the object B that is the value of iv1.

This example may seem pathological since statement (2) could easily be replaced by the statement "(setq iv1 B)", which directly sets iv1 to the object B. However, it illustrates how the side effect of a recursive invocation may alter the values of instance variables. In addition, the recursion may not be as obvious as an object sending a message to itself. It could have involved an object sending a message to another object, which invokes the first object recursively by sending it a message.

When this method is executed on a sequential processor, no error is signaled because the recursion is completely processed before the third statement is executed. However, the computation manager tries to obtain concurrency in the computation by executing the third statement before the results of the second statement are available. It uses rollback to straighten things out when the need arises. Thus, the first time an attempt is made to execute the third statement, the value of iv1 may be 1 and not the object B. When the Lisp runtime system tries to execute this statement with iv1 having the value 1, it signals an error because the number 1 is the wrong type. The computation manager traps this error, and puts the object in a wait state. Eventually, the object A processes the recursive invocation, which causes a rollback, clearing the error condition. The second time statement (3) is executed, iv1 has the value B, and processing proceeds properly.

3.9 GLOBAL VIRTUAL TIME

Global Virtual Time can be used to decide when to commit to exception handling and I/O, as well as allowing old states (and messages) to be discarded. At the present time, support for computing and using Global Virtual Time has not been implemented in the computation manager. This is the next implementation priority.

SECTION 4

COMPILER

4.1 INTRODUCTION

We have designed and implemented a compiler that will translate a useful subset of Common Lisp using the Flavors object oriented extension to code executable by the computation manager. Our analysis of a number of object oriented Common Lisp programs showed that they use only a subset of the language for branching, looping and establishing local environments. Accordingly, we have targeted our initial implementation of the compiler for this subset.

4.2 OVERVIEW OF COMMON LISP

In Common Lisp, programs are written using combinations of *forms*. When a form is evaluated by the Lisp run-time system, it usually returns a value (in some cases multiple values) and, in addition, may cause side effects such as assigning a value to a variable. There are three types of forms: *self-evaluating forms*, *symbol forms*, and *list forms*. A number or a character string is an example of a self-evaluating form because it evaluates to itself. A symbol form evaluates to its value. In this sense, a symbol form plays the same role as a variable in other programming languages. In Lisp, a list is an ordered collection of elements (E1 E2 E3 ... En), where Ei is an element of the list. There are three types of list forms: *special forms*, *function forms*, and *macro forms*. In a list form every element is itself a form.

Common Lisp defines 24 special forms. They are constructs, each with their own syntax, which allow such basic things as assignment, sequential execution, branching, iteration, and local variable definition. The programmer cannot add new special forms. We will discuss some special forms in more detail later.

Common Lisp provides hundreds of functions, and programmers can define additional ones. They are written as (F A1 A2 ... An) where F is the name of a function and Ai is a form. Each Ai is said to be an argument to the function F, and it is said that F is applied to its arguments. When a function form is evaluated, each argument form is evaluated, left to right, and then the function is applied to the returned values. While some functions require a fixed number of arguments, others allow an indefinite number.

For example, the function form (CAR A-LIST) evaluates to the first element of A-LIST. The symbol CAR has a function associated with it that takes one argument which must evaluate to a list. In this example then, A-LIST must be a symbol whose value is a list. To add a number and several symbols' values together, one could write

(+ 1000 COST-OF-MATERIAL COST-OF-LABOR SALES-COMMISSION).

The symbol + refers to the addition function and takes any number of numeric arguments.

The third subtype of list forms is the macro. Common Lisp macros, like those of other languages, aid the programmer by providing high level constructs built from language basics. Also like other languages, macros are replaced textually during evaluation or compilation, as opposed to being linked as a subroutine or function. Common Lisp macros *expand* into a form composed of special forms and functions. Although a macro's expansion may contain other macros, it must ultimately expand to a combination of special forms and/or functions. Common Lisp defines many macros and like functions, the programmer may define additional ones.

Since Common Lisp is such a large language and our goal was to prove concepts, not implement a marketable compiler, it became apparent that supporting a subset of the language would suffice. Clearly, we had to support self-evaluating forms and symbol forms. Macros, as noted above, expand into special forms and functions. Therefore, by having our compiler use the Common Lisp function MACROEXPAND, which expands macros, we did not further consider macros in general. However, the Flavor extension macros DEFFLAVOR, DEFMETHOD, and SEND need special treatment for reasons described later. Since all functions have the same general syntax, that is, a symbol denoting the function followed by arguments all of which are evaluated, we needed only to support one generic function. In order to choose which special forms to support, we conducted an analysis of several object oriented programs. The results indicated that the following eleven special forms would be required: BLOCK, DECLARE, GO, IF, LET, LET*, PROGN, QUOTE, RETURN-FROM, SETQ, and TAGBODY. Any of these special forms may be used directly by a programmer, although there are several that are rarely used by programmers directly because they do not support structured programming practice. Rather, programmers prefer to use macros that support good style. However, these macros use these forms when they are expanded.

These eleven special forms allow many programs to run because they supply the basic programming constructs needed by any programming language. The following is a brief description of the important aspects of these special forms.

(SETQ *variable form*) assigns to *variable* the result of evaluating *form*. The value of *form* is returned as the value of the SETQ special form.

(PROGN *form*$_1$ ... *form*$_N$) provides simple sequencing by evaluating each form, left to right. The value *form*$_N$ is returned as the value of the PROGN special form.

(IF *test-form then-form* [*else-form*]) is the basic branching form. First *test-form* is evaluated. If it evaluates to non-NIL (not false or not empty in the case of data lists), *then-form* is evaluated, and its value is returned as the value of the IF special form. If the value of *test-form* is NIL then *else-form* is evaluated, and its value is returned as the value of the IF special form. If the optional *else-form* is omitted and the value of *test-form* is NIL, NIL is returned as the value of the IF special form.

(QUOTE *object*) simply returns object; it is used to prevent evaluation. This is useful for defining constants. QUOTE is usually seen in its macro form, that is, '*object*.

(LET ((*variable*$_1$ *value*$_1$) ... (*variable*$_L$ *value*$_L$)) *form*$_1$ ... *form*$_N$) establishes a local environment in which the forms are evaluated. First, each of the value forms *value*$_1$ ... *value*$_L$ is evaluated in that order, and their values are saved. Then all the variables *variable*$_1$ ... *variable*$_L$ are bound to the corresponding saved values, conceptually in parallel. Then the forms *form*$_1$ ... *form*$_N$ are evaluated in order. The value of *form*$_N$ is returned as the value of the LET special form.

(LET* ((*variable*$_1$ *value*$_1$) ... (*variable*$_L$ *value*$_L$)) *form*$_1$ ... *form*$_N$) establishes a local environment in which the forms are evaluated. First *value*$_1$ is evaluated and its value is assigned as the value of *variable*$_1$; then *value*$_2$ is evaluated, and its value is assigned as the value of *variable*$_2$; and so on. Then the forms *form*$_1$ ... *form*$_N$ are evaluated in order. The value of *form*$_N$ is returned as the value of the LET* special form. LET* has the same syntax as LET but allows the *value*$_i$ form to refer to the variables *variable*$_1$ ... *variable*$_{i-1}$ previously bound in the LET*.

(TAGBODY {*tag* | *statement*}*) supports iteration. Every element in the body of a tagbody form is either a tag or a statement. A tag is a symbol or an integer. A statement is a form that is neither a symbol nor an integer; usually it is a list form that is evaluated for its side effect. Control is transferred to the part of the body labeled with *this-tag* when a (GO *this-tag*) statement is evaluated. If the last form of the TAGBODY is evaluated, and it is not a GO, NIL is returned. If it is a GO, control is transferred to the tag. Tagbody is rarely used by programmers directly. However, the macros DOTIMES (provides iteration over the non-zero integers) and DOLIST (provides iteration over the elements of a data list) are used frequently by programmers. These macros both use tagbody in their expansion.

(BLOCK *name* $form_1$ ... $form_N$) is used for lexical non-local exits.
Most forms have only one exit and return the value of the last form evaluated. A block form may have more than one exit. *name* must be a symbol. The forms $form_1$ ... $form_N$ are evaluated in order, and the value of $form_N$ is returned as the value of the block unless a (RETURN-FROM *name* [*form*]) with the same name as the encompassing block's *name* is evaluated. In that case, the value of *form* is immediately returned as the value of the block and execution proceeds as if the block had terminated normally. If the *form* is omitted, NIL is returned.

DECLARE {*declaration-specification*}* is used to advise the Common Lisp compiler of ways to make the code more robust or efficient; with one exception — the SPECIAL declaration — it may be ignored. DECLARE is not a form that falls into the category of basic constructs; however, an iteration macro will sometimes use DECLARE in its expansion. Since SPECIAL variables are not supported by the computation manager and efficiency is not a priority, DECLARE was supported in such a way that SPECIAL declarations generate errors and all others are ignored.

Three macro forms are provided by the Flavors extension. These allow the programmer to write object oriented programs. The following is a brief description of the important aspects of these forms.

(DEFFLAVOR flavor-name ($inst\text{-}var_1$ ... $inst\text{-}var_N$) (*other-flavors**) *options**) is used for defining classes of objects used in an application program. Every instance of this flavor has the instance variables $inst\text{-}var_1$ ... $inst\text{-}var_N$. If there are*other-flavors* then the instance variables and methods associated with those flavors are inherited by every instance of *flavor-name*'s class. The *options* that may be specified include *gettable-instance-variables*, *settable-instance-variables*. When these options are specified they direct the Lisp compiler or interpreter to automatically generate methods to access (get) and/or alter (set) the instance variables.

(DEFMETHOD (*flavor-name message-name*) (arg_1 ... arg_M) $form_1$ ... $form_N$) is used to define methods for instances of *flavor-name*'s class. When an instance of this flavor receives the message *message-name* with arguments arg_1 ... arg_M, the forms $form_1$ ... $form_N$ are evaluated in order. The last form evaluated is returned as the value of the method execution.

(SEND *object message-name* $form_1$ ... $form_M$) is used to send a message *message-name* to instance *object*. $form_1$ ... $form_M$ are evaluated in order and their values are bound to the arguments specified in the method definition. The method associated with this message is executed within the scope of the object's instance variables and the arguments specified by the method. The value of the SEND form is the value returned by the method execution.

4.3 REQUIREMENTS

With this overview of the key elements of Common Lisp, it is possible to examine our compiler in some detail. The compiler was needed because the model of execution supports future objects and rolls back when messages are processed out of sequence. To illustrate these ideas, suppose the source code fragment in figure 4-1 is part of a method for an object S.

```
        (PROGN
(1)         (SETQ X 15)
(2)         (SETQ Y (SEND R :m1  X))
(3)         (DECF Z X)
            )
```

Figure 4-1. Code Fragment

In line (1), the variable X is set to the value 15. In line (2), the message :m1, with the value of X as an argument, is sent to the object R; Y is set to the value of the result. In line (3) the variable Z is decremented by the value of X.

Recall from section 3.3 that the computation manager creates a future object to hold the result of the :m1 message before sending the message to the object R. The message sent to the object R contains a pointer to this future object in a special field of the message. While the :m1 message is being processed by R, S's method continues executing; this is the basis of the concurrency obtained by the computation manager. When the :m1 message completes its computation it sends the result to the future object in a :set-value message. A pointer to the future object becomes the value of Y. When the actual value of Y is required the future object is sent a :get-value message to retrieve it.

If the method associated with the :m1 message sent to R were to send a message to the object S, say a message :m2 message, this must be done prior to altering S's instance variables Y and Z. In this example, though, Y may have already been set to a pointer to a future object and Z may have already been decremented before it has been examined. If :m2 altered the value of Z, then the computation done in line (3) will not be done correctly unless the new value is available.

Accordingly, when an object (S) that is currently executing a method receives a message (:m2) with a timestamp indicating recursion, the context associated with the currently executing method is rolled back to the state just after sending the message (:m1) that originally initiated the recursive invocation. This context is then suspended. A new context, associated with the recursive message (:m2) is established. When the recursive context begins executing, the instance variables must be in the state they were in just after the message (:m1) was sent, but before any reply has been received. In particular, this means that if the result of processing the message that originally initiated the recursive invocation (:m1) is to be stored in an instance variable, the value of this variable (Y) must be the value it had just before the message was sent. The recursive context is executed to completion before the suspended context is allowed to continue executing. Because the recursion may alter the state of one or more instance variables (Z), when the suspended context does continue it must resume with the instance variables in the state following the completion of the recursive message.

In the application code, as shown above, a SEND form can occur within another *composite* form (SETQ and PROGN) that contains forms that should be evaluated after the SEND form has been evaluated. When the Lisp runtime system is presented with a composite form it evaluates each *sub*-form in turn before returning control to the computation manager. If the computation manager were to present such a composite form to the Lisp runtime system, it would not be possible to handle recursion properly. To support recursion, a SEND form must be in a separate composite form from the forms that should be evaluated after it.

One purpose of the compiler is to reorganize the application method into a functionally equivalent set of composite forms that can be presented to the Lisp runtime system by the computation manager appropriately. To accomplish this task the source code is broken up into *chunks* of composite forms. Each chunk becomes an element of a compiled method array. The instruction pointer is an index into the array, and the code is executed by evaluating the individual array elements. The computation manager manages the execution of a method by presenting the elements of this array to the Lisp runtime system for evaluation. Disregarding futures, the compiler's output for the above PROGN form code fragment is given in figure 4-2.

```
(SETF (AREF S-M0 2)
    '(PROGN
        (SETQ %INSTR-PTR% 3)
        (SETQ X 15)
        (SETQ %RESULT% (GET-FUTURE))
        (LIST :SEND-REQUEST
           (LIST (SEND R :TW-OBJECT-NAME) 'M1% X)
           (SEND %RESULT% :FUTURE-OBJECT)
           )
        )
    )

(SETF (AREF S-M0 3)
    '(PROGN
        (SETQ %INSTR-PTR% 4)
        (SETQ Y %RESULT%)
        (SETQ %RESULT% (- Z X))
        (SETQ Z %RESULT%)
        )
    )
```

Figure 4-2. Compiled Code Fragment

Since the forms as the user wrote them are destroyed and the new forms are evaluated one chunk at a time, this presents problems for forms such as LET and LET*. As previously mentioned, these forms establish local environments. If in the body of a LET form there is a SEND form, the LET will be broken up into several chunks by the compiler, but each chunk must still be evaluated within the scope of the locally defined variables. To ensure that the evaluation is done correctly, the computation manager's cooperation is required. When the compiler encounters a LET form, code is generated that directs the computation manager to make the local variables available to all chunks that should be evaluated within the scope of the LET (:push-local-variables). After the last form of the LET's body, the compiler inserts a computation manager directive to make these variables unavailable (:pop-local-variables).

The TAGBODY and BLOCK forms are handled similarly because the tags and block names are lexically scoped and the forms' bodies are broken up. In this case, however, no explicit communication with the computation manager is necessary. Instead, the compiler uses a private stack to keep track of each tag, GO, block name, and block RETURN-FROM. A GO or a RETURN-FROM cannot only exit from the encompassing TAGBODY or BLOCK, but can also exit from a LET or LET* form. Because of this, the stack is also used to remember where disestablishment of local environments is required by these non-local exits. This mechanism solves the problem that these control transfers can abnormally terminate the scope of LET, LET*, TAGBODY, and BLOCK.

Breaking up forms into chunks also creates a problem regarding a form's returned value. Most forms return one or more values. Take, for example, the PROGN form. The value returned by the last form evaluated is returned as the value of the PROGN. But, as demonstrated in the code fragment above, the original PROGN becomes distributed among more than one chunk. For this reason, the returning of values must be handled in a way that is unconventional for Lisp. Our implementation uses a reserved instance variable, %RESULT%, defined by the computation manager, to save returned values between – and sometimes during – chunk evaluation. The result of the PROGN form that appeared in the user's original code is the value of (DECF Z X). If that value is used by another form that contains the PROGN it will be made available as the value of %RESULT%.

Another use of the %RESULT% instance variable is to provide a variable to hold a pointer to the future object that will eventually hold the result of (SEND R :m1 X). This pointer becomes the value of Y, but not until the next chunk,because in case of recursion, the value of the variable Y must be available to the recursively called method.

Another purpose of the compiler is to cooperate with the computation manager in the managing of future objects. Future objects are created and managed by the computation manager -- not by the programmer. Since futures are never explicitly declared, the value of almost every Common Lisp symbol appearing as the argument of a function or special form is potentially a pointer to a future object. Whenever such a symbol is encountered, the compiler identifies the context in which the symbol is being used. There are three possibilities: the operation may be performed even if the value of the symbol is a pointer to a future object; an error will be signaled if the value of the symbol is a pointer to a future object; or the operation may return erroneous results if the value of the symbol is a pointer to a future object. The first case occurs when the value of a future symbol is assigned to the value of another symbol or passed as an argument in a SEND. The second case may be implementation dependent, but usually occurs in functions that do type checking, such as arithmetic functions. All other functions fall into the third category, along with the IF special form because of the way the predicate (the test-form) is treated.

Depending upon the way the symbol is used, the compiler either does nothing, lets the error be signaled, or ensures that the symbol is not a future. In the second case, the compiler generates code that sets a computation manager variable %TRAP-FUTURES% to a list of symbols that could be futures. After the variable is set, the computation is attempted. If an error is signaled, the computation manager recovers, uses the list of potential futures to get their values, and then retries the computation. The third case is handled similarly, but instead of setting a variable, the computation manager directive :TEST-FUTURES is generated with the list of potential futures. The computation manager gets the values of all futures in the list before attempting the computation. Next year, we plan to support functions that are defined locally to a method. Consequently, functions will then be able to return futures. The problems introduced by this are being investigated now.

4.4 IMPLEMENTATION

The compiler accomplishes these tasks in two passes. Lisp, being both a recursive language and defined recursively, is also conveniently compiled using a recursive program. This is the approach taken by our compiler. The result is a compact compiler with just over 800 lines of code. In the first pass, all macros except the Flavor extension macros SEND, DEFMETHOD, and DEFFLAVOR are expanded into their special forms and function definitions.

As shown above in the sample compiled code, the SEND macro is translated by our compiler into a form involving a future object and a form containing the computation manager directive :send-request. In most implementations of Flavors, DEFMETHOD is compiled into a Common Lisp function. Our compiler creates an array whose elements are Lisp forms. The computation manager uses this compiled method array to manage the execution of the method. In addition, the method's name and formal parameters are saved for use in the associated DEFFLAVOR, while the body is processed like a PROGN form.

DEFFLAVOR is used to define classes in our system also. However, the application program's DEFFLAVOR definitions must be modified by the compiler to suit our purposes. For example, an instance variable is added for each method, so that the code array can be found and the method's arguments known. Also, a Flavor definition may specify that methods should be automatically generated that access and/or alter the instance variables (recall the *options* specifier of DEFFLAVOR described previously). If the application DEFFLAVOR specifies any of these options, our compiler creates compiled method arrays and names them according to Flavor conventions. They, like user defined method definitions, are then processed during pass 2.

Each DEFMETHOD macro is converted into a compiled method array that the computation manager uses to manage the execution of the method. Specifically, the compiler creates an array, each element of the array is assigned a chunk of code. These chunks of code consist of up to three components: flow control, user code, and computation manager directives. A method is executed by evaluating the contents of certain elements of the code array. Flow control determines which elements are evaluated and in what order. When a method is invoked, an instruction pointer is set to the first element, and its associated code is evaluated. Part of the evaluated code must set the instruction pointer to the next element to be evaluated. An important point is that the evaluation of a code chunk can be viewed as an atomic action. That is, the entire chunk is evaluated by the Lisp runtime system before control is returned to the computation manager.

The user code component, with perhaps a computation manager directive, is simply a representation of the user's source code. Although the structure and Lisp forms are different from the source, they accomplish the same thing. A directive, if present, must appear at the end of a chunk. This implies that only one per chunk is allowed. These directives instruct the computation manager to perform such actions as sending messages, controlling local variables, and notifying the object associated with a context that the execution of the context is complete. These operations are performed under the control of the computation manager by the context object methods associated with the messages :SEND-REQUEST, :PUSH-LOCALS, :POP-LOCALS, and :SEND-DONE described in section 3.2.

4.5 AN EXAMPLE

The following example should clarify the foregoing discussion. Because of the compiler's voluminous output, the example has been kept short. Given the Flavor and method definitions of figure 4-3, figure 4-4 shows the output of pass 1 when applied to the body of the DOLIST macro, and figure 4-5 shows the code the compiler yields.

In figure 4-3, the Flavor definition specified that the instance variable(s) must be "gettable," that is, a method should be automatically generated that returns the value of the instance variables. Therefore, at the top of figure 4-5, there is a compiled method to "get" or access the instance variable MESSAGES-SENT. The compiled method OBJECT-1-M1 is then shown which corresponds to the user's method. Near the end of figure 4-5 is the context object Flavor definition based on the user's Flavor definition. At the bottom of the figure is an uncompiled method generated by the compiler for examining objects controlled by the computation manager.

It can be seen in figure 4-4 that any form requiring pass 2 processing has been marked with *PASS2*. Also, the SEND macro has been replaced by a computation manager directive referencing the context object method :SEND-REQUEST. Finally, the symbols THIS-OBJECT and MESSAGES-SENT have been flagged as being potential Future objects.

```
(defflavor object-1
   ((messages-sent 0))
   ()
   :gettable-instance-variables
   )

(defmethod (object-1 :m1) (object-list)
   (dolist (this-object object-list)
      (send this-object :m2)
      (setq messages-sent (+ messages-sent 1))
      )
   )
```

Figure 4-3. Source Code

```
((*PASS2* PROGN
     (*PASS2* TEST-FUTURES (THIS-OBJECT))
     (SETQ %RESULT% (GET-FUTURE))
     (*PASS2* SEND-FLAG
          LIST
          :SEND-REQUEST
          (LIST (SEND THIS-OBJECT :TW-OBJECT-NAME) 'M2%)
          (SEND %RESULT% :FUTURE-OBJECT)
          )
     )
 (*PASS2* PROGN
     (*PASS2* PROGN
          (*PASS2* TRAP-FUTURES (MESSAGES-SENT))
          (SETQ %RESULT% (+ MESSAGES-SENT 1))
          )
     (SETQ MESSAGES-SENT %RESULT%)
     )
)
```

Figure 4-4. Pass 1 Output of DOLIST Body

Figure 4-5. Compiled Methods and Context Definitions

```
(DEFFLAVOR OBJECT-1
    (TW-OBJECT-NAME)
    NIL
    :SETTABLE-INSTANCE-VARIABLES
    )


(DEFPARAMETER OBJECT-1-MESSAGES-SENT (MAKE-ARRAY 3))

(SETF (AREF OBJECT-1-MESSAGES-SENT 0)
   '(PROGN
        (SETQ %INSTR-PTR% 1)
        (SETQ %ADR-POP-LOCALS% NIL)
        (SETQ %RESULT% MESSAGES-SENT)
        (LIST :SEND-REQUEST
            (LIST %REQUESTER% 'SET-VALUE (LIST 'QUOTE %RESULT%)) NIL)
        ))
(SETF (AREF OBJECT-1-MESSAGES-SENT 1)
   '(PROGN
        (SETQ %INSTR-PTR% 2)
        '(:SEND-DONE)
        ))
(SETF (AREF OBJECT-1-MESSAGES-SENT 2)
   '(PROGN
        (FORMAT T "~s method completely executed instr-ptr = ~s~%" NAME %INSTR-PTR%)
        (INCF %INSTR-PTR% 3)
        ))


(DEFPARAMETER OBJECT-1-M1 (MAKE-ARRAY 21))
(SETF (AREF OBJECT-1-M1 0)
   '(PROGN
        (SETQ %INSTR-PTR% 1)
        (SETQ %ADR-POP-LOCALS% NIL)
        '(:PUSH-LOCALS ((%COMPILER2 NIL)))
        ))
(SETF (AREF OBJECT-1-M1 1)
   '(PROGN
        (SETQ %INSTR-PTR% 2)
        '(:PUSH-LOCALS ((%COMPILER0 NIL)))
        ))
```

```
(SETF (AREF OBJECT-1-M1 2)
    '(PROGN
        (SETQ %INSTR-PTR% 3)
        (SETQ %COMPILER0 OBJECT-LIST)
        '(:PUSH-LOCALS ((THIS-OBJECT NIL)))
        ))
(SETF (AREF OBJECT-1-M1 3)
    '(PROGN
        (SETQ %INSTR-PTR% 4)
        (SETQ %TRAP-FUTURES% '(%COMPILER0))
        (SETQ %RESULT% (CAR %COMPILER0))
        (SETQ THIS-OBJECT %RESULT%)
        NIL
        ))
(SETF (AREF OBJECT-1-M1 4)
    '(PROGN
        (SETQ %INSTR-PTR% 5)
        '(:TEST-FUTURES (%COMPILER0))
        ))
(SETF (AREF OBJECT-1-M1 5)
    '(PROGN
        (SETQ %INSTR-PTR% 6)
        (SETQ %RESULT% (NULL %COMPILER0))
        (SETQ %COMPILER2 %RESULT%)
        '(:TEST-FUTURES (%COMPILER2))
        ))
(SETF (AREF OBJECT-1-M1 6)
    '(PROGN
        (SETQ %INSTR-PTR% (IF %COMPILER2 7 8))
        ))
(SETF (AREF OBJECT-1-M1 7)
    '(PROGN
        (PROGN
            (SETQ %RESULT% NIL))
        (SETQ %NUM-POP-LOCALS% 0)
        (SETQ %RETURN-INSTR-PTR% %ADR-POP-LOCALS%)
        (SETQ %INSTR-PTR% 15)
        ))
(SETF (AREF OBJECT-1-M1 8)
    '(PROGN
        (SETQ %INSTR-PTR% 9)
        (SETQ %RESULT% NIL)
        ))
(SETF (AREF OBJECT-1-M1 9)
    '(PROGN
        (SETQ %INSTR-PTR% 10)
        '(:TEST-FUTURES (THIS-OBJECT))
        ))
```

```
(SETF (AREF OBJECT-1-M1 10)
    '(PROGN
        (SETQ %INSTR-PTR% 11)
        (SETQ %RESULT% (GET-FUTURE))
        (LIST :SEND-REQUEST
          (LIST (SEND THIS-OBJECT :TW-OBJECT-NAME) 'M2%)
           (SEND %RESULT% :FUTURE-OBJECT))
        ))
(SETF (AREF OBJECT-1-M1 11)
    '(PROGN
        (SETQ %INSTR-PTR% 12)
        (SETQ %TRAP-FUTURES% '(MESSAGES-SENT))
        (SETQ %RESULT% (+ MESSAGES-SENT 1))
        (SETQ MESSAGES-SENT %RESULT%)
        (SETQ %RESULT% NIL)
        ))
(SETF (AREF OBJECT-1-M1 12)
    '(PROGN
        (SETQ %INSTR-PTR% 13)
        (SETQ %TRAP-FUTURES% '(%COMPILER0))
        (SETQ %RESULT% (CDR %COMPILER0))
        (SETQ %COMPILER0 %RESULT%)
        ))
(SETF (AREF OBJECT-1-M1 13)
    '(PROGN
        (SETQ %TRAP-FUTURES% '(%COMPILER0))
        (SETQ %RESULT% (CAR %COMPILER0))
        (SETQ THIS-OBJECT %RESULT%)
        (SETQ %NUM-POP-LOCALS% 0)
        (SETQ %RETURN-INSTR-PTR% %ADR-POP-LOCALS%)
        (SETQ %INSTR-PTR% 4)
        ))
(SETF (AREF OBJECT-1-M1 14)
    '(PROGN
        (SETQ %INSTR-PTR% 15)
        (SETQ %RESULT% NIL)
        ))
(SETF (AREF OBJECT-1-M1 15)
    '(PROGN
        (SETQ %INSTR-PTR% 16)
        '(:POP-LOCALS)
        ))
(SETF (AREF OBJECT-1-M1 16)
    '(PROGN
        (SETQ %INSTR-PTR% 17)
        '(:POP-LOCALS)
        ))
(SETF (AREF OBJECT-1-M1 17)
    '(PROGN
        (SETQ %INSTR-PTR% 18)
        '(:POP-LOCALS)
        ))
```

```
(SETF (AREF OBJECT-1-M1 18)
    '(PROGN
        (SETQ %INSTR-PTR% 19)
        (LIST :SEND-REQUEST
            (LIST %REQUESTER% 'SET-VALUE (LIST 'QUOTE %RESULT%)) NIL)
        ))
(SETF (AREF OBJECT-1-M1 19)
    '(PROGN
        (SETQ %INSTR-PTR% 20)
        '(:SEND-DONE)
        ))
(SETF (AREF OBJECT-1-M1 20)
    '(PROGN
        (FORMAT T "~s method completely executed instr-ptr = ~s~%" NAME %INSTR-PTR%)
        (INCF %INSTR-PTR% 3)
        ))


(DEFFLAVOR %OBJECT-1%
     ((M1% (MAKE-INSTANCE '%ENVIRONMENT-CLASS%
                          :CODE OBJECT-1-M1
                          :ARGS '(OBJECT-LIST)))
      (%REAL-INST-VARS% '(MESSAGES-SENT))
      (MESSAGES-SENT% (MAKE-INSTANCE '%ENVIRONMENT-CLASS%
                                     :CODE OBJECT-1-MESSAGES-SENT
                                     :ARGS (QUOTE NIL)))
      (MESSAGES-SENT 0)
      )
     NIL
     (:GETTABLE-INSTANCE-VARIABLES %REAL-INST-VARS%)
     )


(DEFMETHOD (%OBJECT-1% :EXPLAIN) (&OPTIONAL (N-BLANKS 0))
     (FORMAT T "~vt ~a = ~s~%" N-BLANKS "MESSAGES-SENT" MESSAGES-SENT)
     )
```

SECTION 5

RESOURCE MANAGEMENT ALGORITHMS AND SIMULATION

5.1 INTRODUCTION

We expect that objects are dynamically created and discarded (when no longer needed) as a program executes. We think of objects as representing units of work assigned to processors. Memory management must provide a means of finding processors with sufficient free memory to store new objects. Processor management must provide a means of dynamically balancing the load on processors by distributing objec:s in a relatively equitable manner.

Together with the Fault Tolerant Storage Reclamation project, we have continued the investigation of fault tolerant resource management begun in fiscal year 1987 under the Future Generation Computer Architectures project. We have developed a resource management strategy that meets these requirements. One feature of the scheme is that objects are distributed among processors in an equitable manner when they are created. Another feature is that objects are dynamically redistributed among processors to maintain a relatively equitable distribution.

The scheme uses a control hierarchy (figure 5-1). At each level of that hierarchy, decisions about object creation and redistribution are made locally. Each level coordinates decisions that cannot be made without crossing the boundaries of local regions at the next lower level. All decisions are made using inexact information about current resource use. As a result, resource management algorithms require less overhead to be able to tolerate hardware failures.

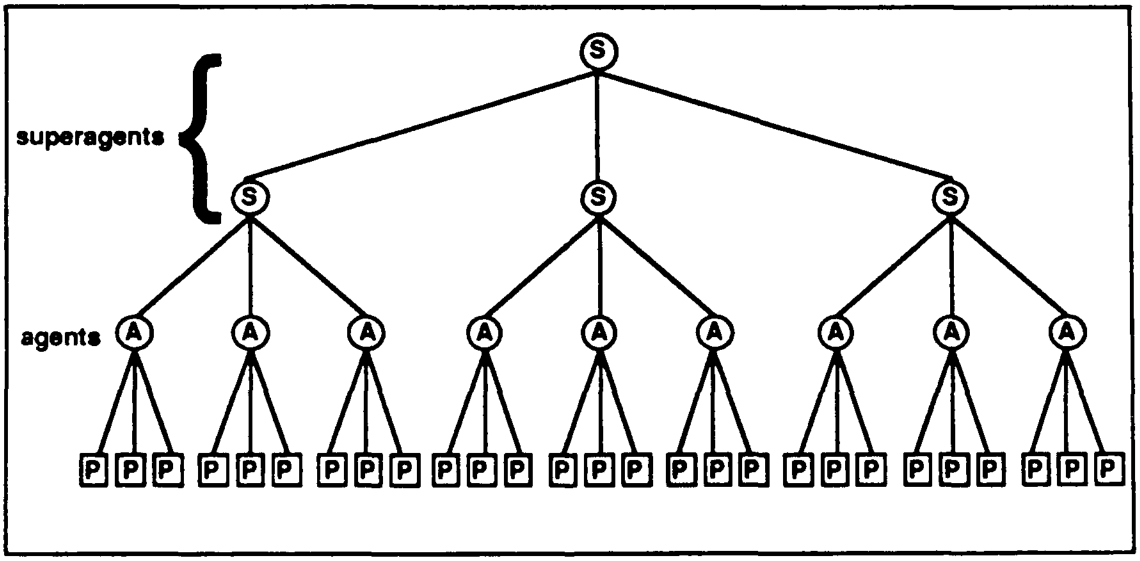


Figure 5-1. Resouce Management Hierarchy

The scheme is based on the use of resource *agents* -- operating system servers distributed throughout the system. Each agent manages memory allocation for the processors in its local communication neighborhood. Whenever the memory use in a processor crosses a predetermined level (e.g., 10%, 20%, etc.), the processor reports to its local agent. Whenever the net memory use in an agent's neighborhood crosses a predetermined level (e.g., 10%, 20%, etc.), the agent reports to a *super*agent -- another operating system server that oversees activity in several agents' neighborhoods. In a similar manner, superagents themselves report to other superagents above them. In this sense, a superagent's neighborhood is the union of its inferiors' neighborhoods.

5.2 ALLOCATION

Suppose an object needs to create an instance of a new object. Without specific information about which objects will want to communicate with a new object, it is reasonable to create the object in a processor that is as close as possible to the one containing the object that requests the creation. The requesting object is initially the only one that knows about the existence of the new object, and, therefore, the only one able to communicate with it.

A processor sends allocation requests for a specific number of memory units to its local agent (figure 5-2). That agent assigns each request to whichever processor in its neighborhood it deems most appropriate; if there is none, it forwards the request to its superagent. If the superagent has another inferior that may be able to satisfy the request, it forwards the request to that inferior; otherwise, it forwards the request to its own superagent.

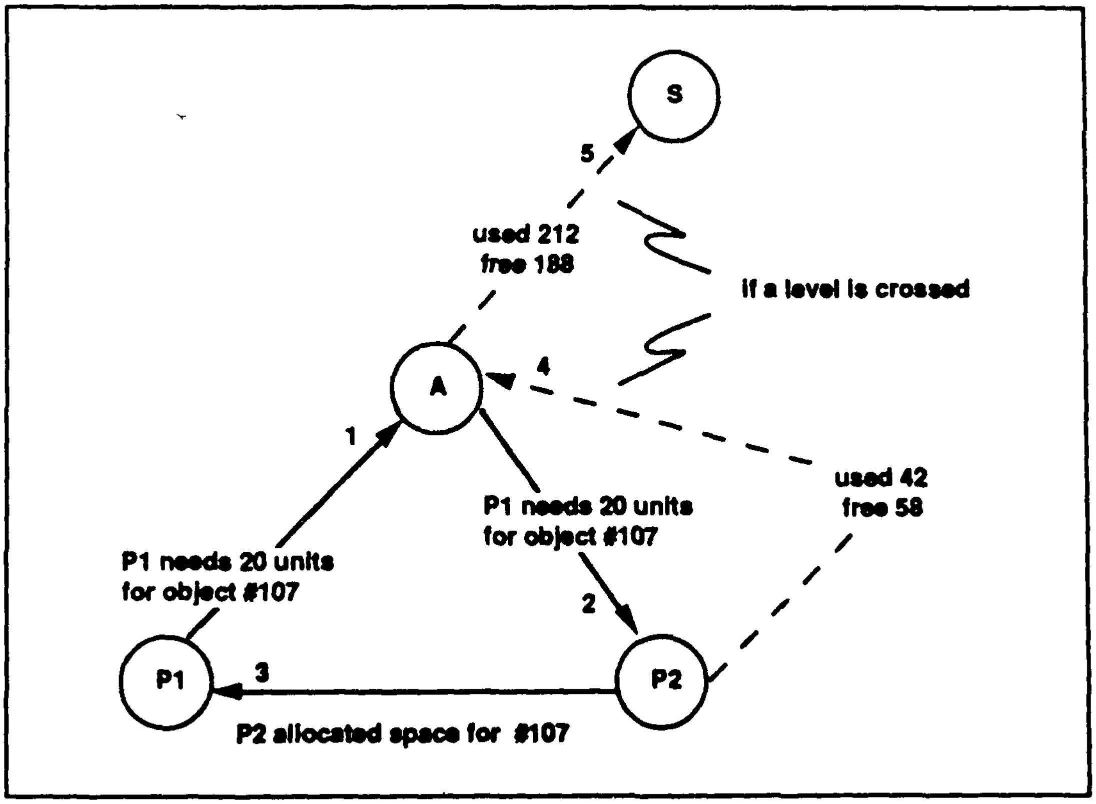


Figure 5-2. Allocation Request

If the allocation of an object on a processor causes the memory use in the processor to cross a predetermined level (for example, from below 40% to above 40%), the processor sends an *update* message to its agent to report its current level, the amount of memory in use and the amount free (for example, the processor reports 42 units of memory in use, 58 units free). The agent uses this information as the basis for selecting an appropriate processor as a candidate for future allocations.

An agent only has approximate knowledge of the memory use on the processors in its neighborhood; processors report only when they cross predetermined levels. Furthermore, these messages may not arrive in a timely manner; in addition to the possibility of lost messages, we make no assumptions about the order of message delivery. Thus, an agent may attempt to assign an allocation to a processor that cannot honor it because it does not have enough free memory. When this happens, the processor sends a message to the agent refusing the allocation; this message also contains information on the processor's current memory use. When an agent receives an allocation refusal, it may attempt to assign the allocation to another processor in its neighborhood; if it does not have a processor it deems appropriate, it passes the allocation request to its superagent.

Every time an agent receives updated information from one of its processors, the agent calculates the net resource use in its neighborhood. If a predetermined level has been crossed, the agent sends an *update* message to its superagent. Similarly, every time a superagent receives an *update* message from one of its agents, the superagent calculates the net resource use in its neighborhood. If a predetermined level has been crossed, the superagent sends an *update* message to its superior (that is, its superagent).

Since we make no assumptions about the order of message delivery, every processor, agent, and superagent numbers the *update* messages it sends. This allows a superior to ignore earlier *update* messages that are received out of sequence.

5.3 DEALLOCATION

When the garbage collector identifies an object as no longer needed by the application, the processor that is storing the object deallocates the object's space. If this causes the memory use in the processor to cross a predetermined level (for example, from above 40% to below 40%), the processor sends an *update* message to its agent (for example, the processor reports that its current level is 35 units of memory in use, 65 units free). We have already described what happens when an agent receives an *update* message.

5.4 REDISTRIBUTION

For this scheme to work efficiently, an agent should be able to find a processor in its own neighborhood that can satisfy an allocation request. To this end, each superagent employs a redistribution strategy to move objects from higher-density areas to lower-density areas within the superagent's purview.

When a superagent receives an *update* message, it checks to see if the disparity in resource use among its inferiors' neighborhoods exceeds a predetermined threshold. If it does, the superagent directs the inferior with the highest level (by sending it a *reduce* message) to move objects to the neighborhoods of those inferiors with which it differs by the threshold. The *reduce* message contains a target level the inferior is to achieve, so that its disparity with the superagent's other inferiors will be below the threshold. For example, suppose memory use levels are reported each time an inferior crosses a multiple of 10%, and the disparity threshold is 20%. Suppose superagent S has four inferiors in its neighborhood S0, S1, S2, and S3 with memory use levels of 32%, 44%, 47%, and 31%, respectively. Now suppose S receives an update memory use message from S2 that indicates that its current memory use is 58%. At this point the disparity between S2 and S0 and the disparity between S2 and S3 are both above the threshold. S sends a message to S2 to reduce its memory use level to below 50% by sending objects from its neighborhood to S0's and S3's neighborhoods (figure 5-3). Agents use a similar strategy to redistribute objects among the processors in their neighborhoods.

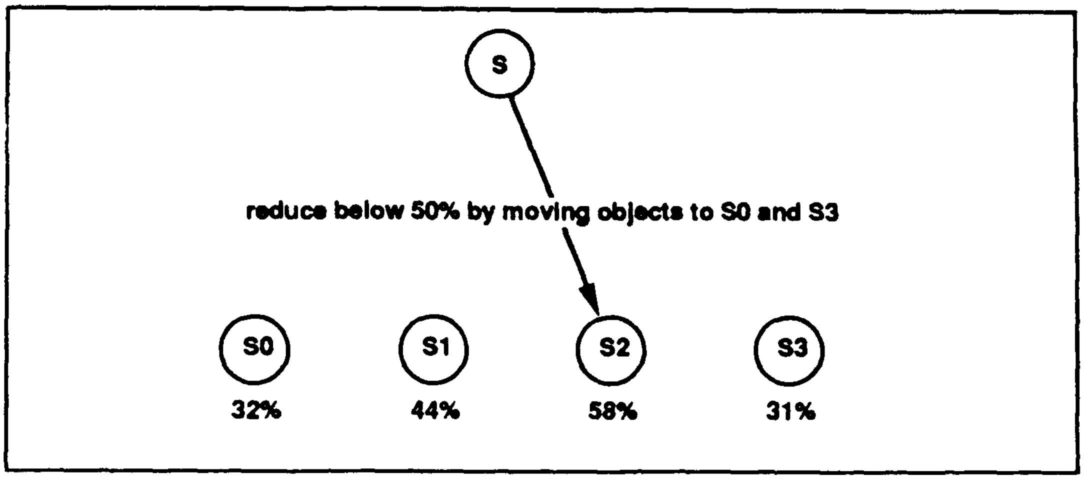


Figure 5-3. Initiate Redistribution

When an inferior is told to move objects to some other neighborhood, it forwards that directive to its own inferiors. The directive moves down the hierarchy until it reaches the agents. Agents that are told to move objects tell their processors to select objects and move them to the destination neighborhoods. When an object is moved, the same messages are used to find space for the object as when an object is created (figure 5-4).

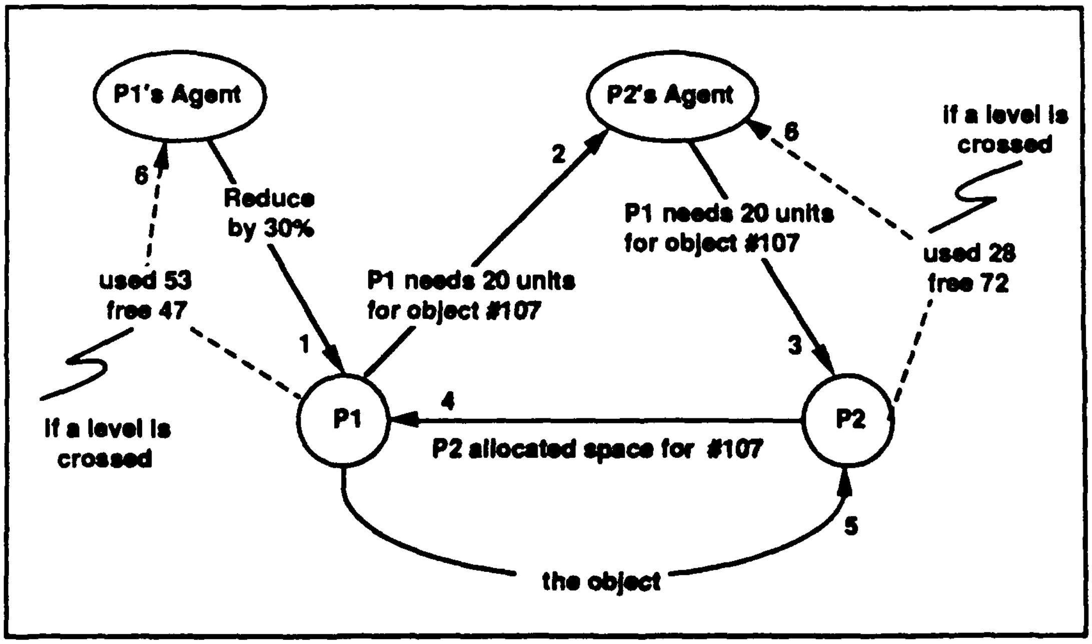


Figure 5-4. Move Object Request

A processor continues to move objects until it reaches its target level; whenever it crosses a memory use level it sends an *update* message to its agent. When the net memory use level in an agent's neighborhood crosses a memory use level, it sends an *update* message to its superagent. When the net memory use level in a superagent's neighborhood crosses a memory use level, it sends an *update* message to its superagent. In this way updated resource use information is eventually reported up the hierarchy until the superior that originally initiated the redistribution is satisfied that a reasonable balance has been achieved among its inferiors.

Whenever the superagent that initiated a redistribution receives an *update* message, it reevaluates its redistribution decision. The superagent may decide that a different target level should be achieved, or that objects should be moved to different neighborhoods, or that redistribution should be stopped; if it does, it sends a new directive to its inferiors. Since we make no assumptions about the order of message delivery, every superagent numbers the *reduce* messages it sends. This allows an inferior to ignore earlier *reduce* messages that are received out of sequence.

Any superagent at any level in the hierarchy may decide to redistribute objects to balance the load in its neighborhood. Directives to redistribute from higher-level superagents supersede those of lower-level superagents. Thus, if a superagent that is currently redistributing objects within its own neighborhood receives a *reduce* message from its superior, it sends a new directive to its inferiors. When an inferior receives a new directive from its superior, the old directive is discarded and the new directive becomes the operative one.

5.5 SECONDARY STORAGE

We assume that some agents have secondary storage device controllers in their neighborhoods. In addition to redistributing objects, each superagent employs a strategy for moving objects to secondary storage when primary storage begins to saturate. When a superagent receives updated resource information from its inferiors, it checks to see if the total resource use among its inferiors' neighborhoods exceeds a predetermined threshold. If it does, and if the superagent has not received a command to move objects to other neighborhoods, and if there are secondary storage controllers in its neighborhood, it directs those inferiors that have secondary storage controllers in their neighborhoods to move objects to secondary storage by sending them a *ss-reduce* message. The *ss-reduce* message contains a target level the inferior is to achieve.

When an inferior is told to move objects to secondary storage, it forwards that command to those of its own inferiors that have secondary storage controllers in their neighborhoods. The command moves down the hierarchy until it reaches the agents. Agents that are told to move objects to secondary storage tell their processors to select objects and move them to the secondary storage controllers.

A processor continues to move objects from its memory to secondary storage until it has reached (or exceeded) the resource use level it was directed to achieve. As a processor crosses a resource use level it reports its updated resource use information to its agent. Similarly, an agent continues to direct its processors to move objects until it has reached (or exceeded) the resource use level it was directed to achieve. As an agent crosses a resource use level, it reports its neighborhood's updated resource use level to its superior. In this way, updated resource use information is eventually reported up the hierarchy until the superior that originally initiated the movement of objects to secondary storage is satisfied that a sufficient amount of free primary storage is now available in its neighborhood. While secondary storage movement is in progress, redistribution is also being performed to maintain a reasonable balance of resource use.

Each time the superior that initiated a movement of objects to secondary storage receives an *update* message, it reevaluates its secondary storage movement decision. The superagent may decide that a different target level should be achieved or that movement to secondary storage should be stopped; if it does, it sends a new directive to its inferiors. Since we make no assumptions about the order of message delivery, every superagent numbers the *ss-reduce* messages it sends. This allows an inferior to ignore earlier *ss-reduce* messages that are received out of sequence.

Any superagent at any level may decide to move objects to secondary storage to free primary memory in its neighborhood. Directives to move objects to secondary storage from higher-level superagents supersede those of lower level superagents. Thus, if a superagent that is currently directing the movement of objects to secondary storage within its own neighborhood receives an ss-*reduce* message from its superior, it sends a new directive to its inferiors. When an inferior receives a new directive from its superior, the old directive is discarded and the new directive becomes the operative one.

5.6 FAULT TOLERANCE

Reliability is especially important in a system as large as a million processors. Given the size and complexity of such a system, individually reliable components are not sufficient to guarantee the reliability of the system as a whole. Roughly speaking, a system comprised of millions of components, each with a mean-time-between-failures (MTBF) of even hundreds of thousands of hours, can expect to have a net MTBF of a few minutes. The development of large-scale multiprocessors underscores the need for hardware fault tolerance.

The only kind of hardware failure we consider is catastrophic processor and communication link failure. We assume that known techniques -- like the use of error-detecting and correcting codes, comparison of duplicated operations, and protocol monitoring -- are used to ensure that processors that function do so in a fault-free manner and that communication between processors is reliable [Anderson85, Bernstein88].

We also assume the connectivity among processors is such that there is more than one path between any pair of processors, and we assume the hardware or software that makes message-routing decisions is sophisticated enough to make use of alternate paths to bypass failed processors or communication links [Chow87]. As time passes and the number of failures throughout the system increases, there will come a time when there is no path between a given pair of processors. This violates our assumption about connectivity, and must not be allowed to occur. It is essential, therefore, that failed hardware can be replaced with new hardware while the system is running, and that the operating system can recognize new hardware and assimilate it into the running system.

When a processor fails, the objects and messages that are in that processor are lost. Some of those messages may be operating system messages, and others application messages. In addition, we lose any operating system services that were being provided by the dead processor. All subsequent messages to the processor will be returned to the sender as undeliverable.

We only require that the system tolerate isolated hardware failures. By isolated, we mean there must be a sufficient amount of time following a failure for the system to heal itself. There may be other failures in other parts of the system, but those parts of the system that contribute to restoring functionality after the first failure must not fail themselves, at least until they have completed their failure-recovery tasks.

Our view of fault tolerance is an obviously probabilistic view -- as must be any view, whether or not it is stated as such. When we say the system can tolerate isolated failures as we have defined them, we are really saying that the probability of system failure is the probability of two failures occurring in related parts of the system within a given time window.

5.6.1 Lost Messages

Resource managemenι must tolerate lost messages. If a processor sends an allocation request to its agent and does not receive a response within a predetermined amount of time, it assumes that a message was lost. When this happens, the processor sends the request message to its agent again. This is repeated until the processor receives a response.

If no message was lost (requests that have to travel far outside the processor's neighborhood may take longer than the time-out interval to be satisfied), the processor may receive a response to its first request after it has repeated that request to its agent. The space identified by the first response received is used for the object; subsequent responses are ignored -- they represent space that is allocated, but not used. Eventually, the garbage collector recognizes that this space is not being used, and deallocates it. We discuss ways to reduce the number of times lost messages cause redundant memory allocation below.

Resource management must tolerate lost *update* messages. If the lost message was from a processor to an agent, this may cause the agent to have a poor estimate of the processor's current resource use. An underestimate may cause the agent to assign an allccation to a processor that cannot satisfy it, because the processor does not have enough free memory. We addressed this possibility above. An overestimate may cause the agent to stop assigning allocations to a processor that has available memory. To recover from this situation, an agent that has not received an *update* message from a processor in a sufficiently long time sends a message to that processor requesting an update. (This message also allows the agent to check that the processor has not failed.) Similar statements apply to lost *update* messages from agents to superagents, and from superagents to their superiors.

If an *update* message from a processor to an agent is lost, the agent may decide to redistribute objects in its neighborhood. Depending on whether the agent underestimates or overestimates the processor's resource use, it may direct that objects be moved to or from that processor. The amount of incorrect redistribution that may take place depends on the spacing between levels at which processors send *update* messages. When objects are being moved to or from a processor, at most one level's worth of objects can be moved before the processor sends an *update* message to its agent. Thus, the more closely spaced are the levels at which *update* messages are sent, the more quickly the agent can reevaluate its decision to redistribute objects. Similar statements apply to lost *update* messages from agents to superagents, and from superagents to their superiors.

Finally, resource management must tolerate lost *reduce* and *ss-reduce* messages. To address this possibility, an agent or superagent periodically resends *reduce* or *ss-reduce* messages.

5.6.2 Lost Processors, Agents, and Superagents

Resource management must tolerate lost processors, agents, and superagents. If a message from an agent to a processor is returned as undeliverable (because the processor has failed), the agent removes that processor from the list of processors it is managing, recalculates the net resource use in its neighborhood, and sends an *update* message to its superagent.

If a message from a processor to an agent is returned as undeliverable (because the agent has failed), the processor must know of another processor to which it can send its allocation requests. Thus, every agent and superagent is assigned a backup. Every agent knows the locations of its own backup and its superagent's backup. Every superagent knows the locations of its own backup, superior's backup, and inferiors' backups. An agent's backup knows its agent, superagent, superagent's backup, and which processors are in its neighborhood, but nothing about the resource use in those processors. A superagent's backup knows its own superagent, superior, superior's backup, inferiors' backups and which inferiors are in its neighborhood, but nothing about the resource use in the inferiors' neighborhoods (figure 5-5).

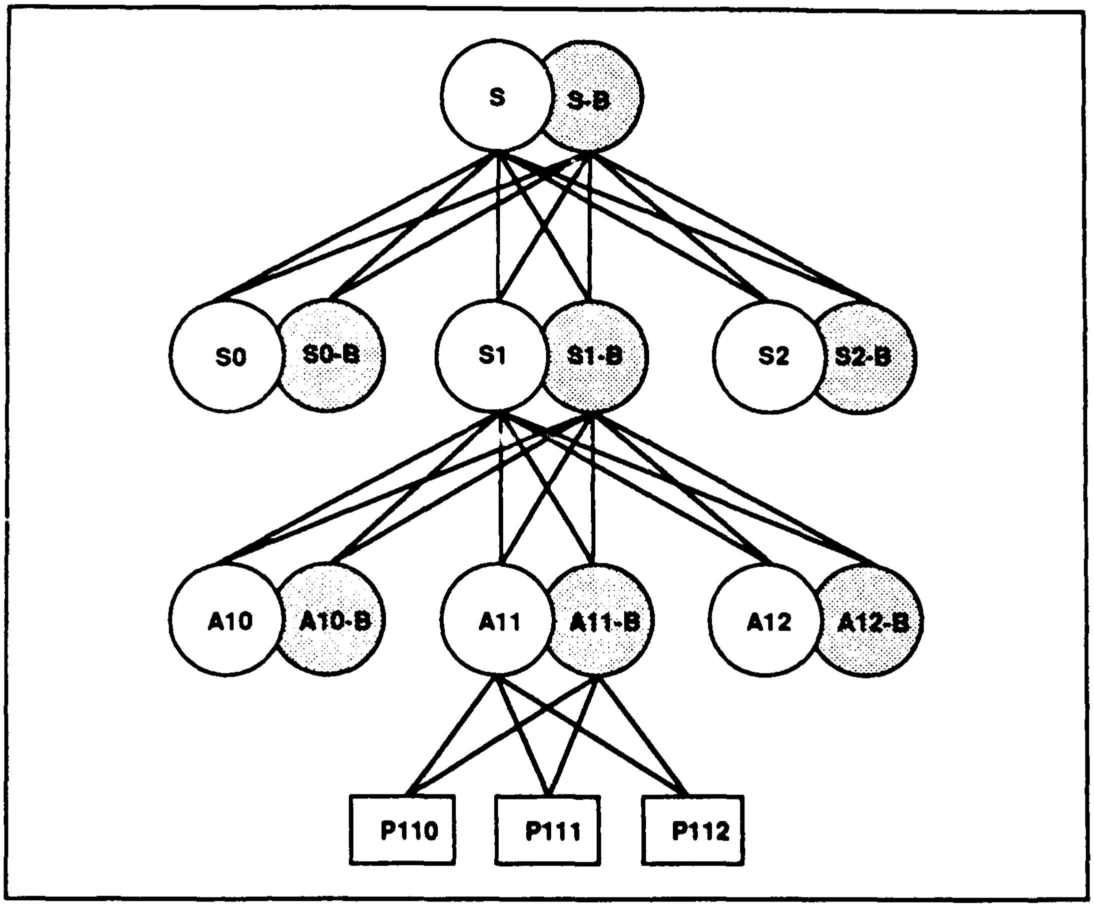


Figure 5-5. Superagents, Agents and Backups

Agents, superagents, and their backups communicate with each other occasionally to make sure they are still functioning. When an agent or superagent fails, its backup is notified. The backup performs the following: (1) it makes a new backup to take its place; (2) it informs its superior and inferiors of its new status, and the location of the new backup; and (3) it receives status information from its inferiors that enables it to continue managing resource use in its neighborhood (figure 5-6). When a backup fails, the corresponding agent or superagent creates a new backup to replace the failed one.

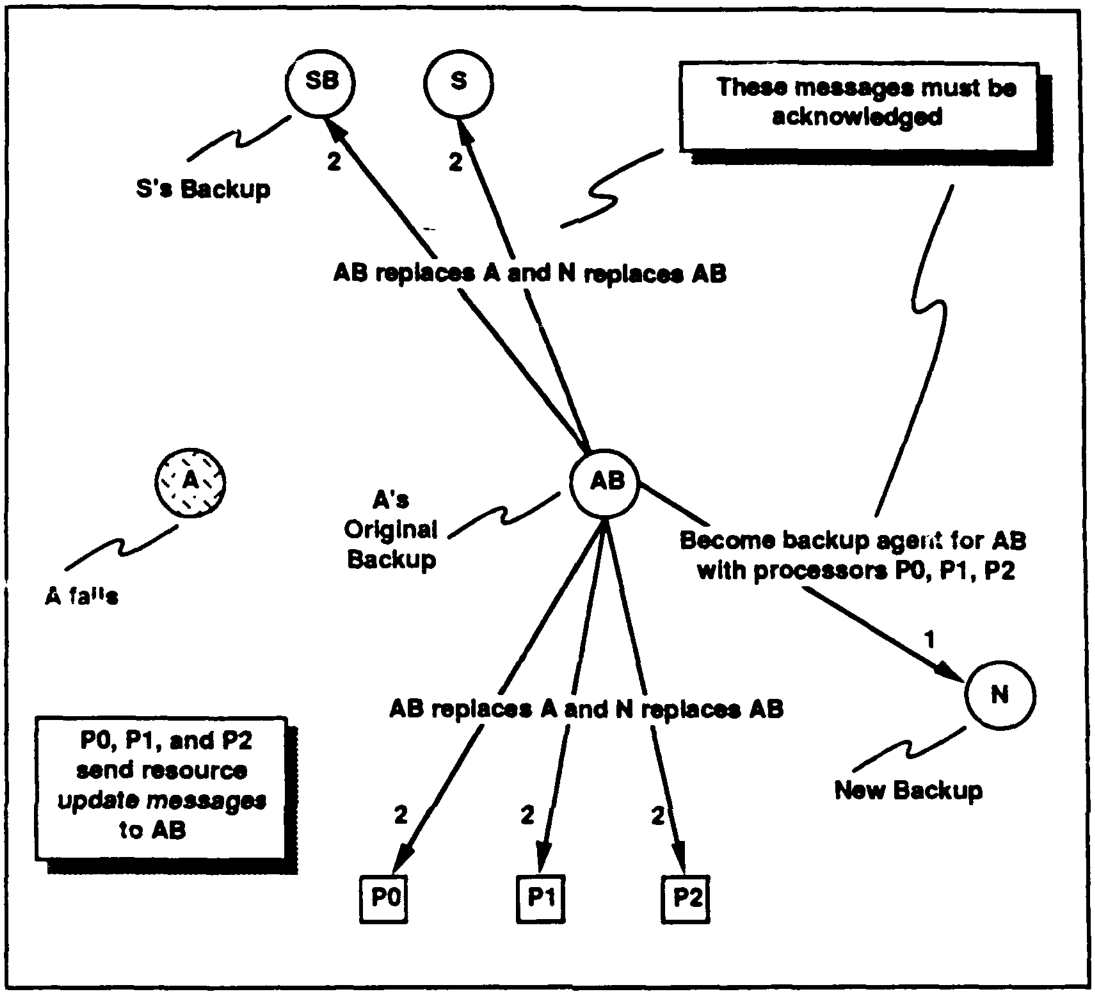

SB
S
These messages must be acknowledged
S's Backup
2
2
AB replaces A and N replaces AB
A
AB
A's Original Backup
A fa''s
Become backup agent for AB with processors P0, P1, P2
1
N
AB replaces A and N replaces AB
P0, P1, and P2 send resource update messages to AB
New Backup
2
2
2
P0
P1
P2

Figure 5-6. Creating a New Backup Agent

Agents and superagents can be located on any processors. Ideally, an agent should be as close as possible (in terms of communication distance) to the processors in its neighborhood. Likewise, a superagent should be as close as possible to its inferiors. A backup should be located in the same neighborhood as the agent or superagent it backs up. Superagents and their backups should be distributed as uniformly as possible throughout the system.

Consider the situation when an agent fails and no functioning processor other than the agent's backup exists in the agent's neighborhood. In this case it is not possible to establish a new backup for the agent in its own neighborhood. Think of what this means: the agent's processor is the only one left functioning in the agent's neighborhood. We believe the reasonable thing to do in such an extreme case is for the agent to shut itself down. This means its processor hands over responsibility for application objects and operating system services to other processors and goes into a restricted mode of operation in which, perhaps, it only participates in message passing activities.

5.7 ALGORITHM CONCLUDING REMARKS

Resource management decisions are made at the local level whenever possible. Superagents only become involved in resource management when coordination is needed among lower level neighborhoods. This reduces communication and decision-making bottlenecks at higher levels of the superagent hierarchy. The scheme is, therefore, more amenable to implementation in larger systems.

As we have described, agents use inexact information about resource use in their neighborhoods to satisfy allocation requests, and backups have no information about resource use until they take over for failed agents. This requires less message overhead than if agents and their backups are given exact information. However, the less frequently processors report status changes, or the longer those messages take to reach agents, the more out-of-date an agent's information can become, and the more likely it is to assign a given request to a processor that cannot satisfy it, or to assign requests, so that resource use is not as uniform as possible throughout the neighborhood. If this happens frequently, it may be more efficient to lower the thresholds or to consider giving agents (and even their backups) perfect knowledge of memory use.

We promised to discuss ways to reduce the number of times lost messages cause redundant memory allocation. Redundant allocation can be avoided if an agent can remember which processor it assigns to each allocation request, and a processor can remember which allocations it has been assigned. However, agents and allocating processors cannot remember how allocation requests are satisfied forever. One solution is to tell them when they can forget how a given request was satisfied. If the rate of allocation requests is high, or if it takes a long time for agents and processors to be told they can forget a particular request, then it may be impossible for them to remember everything necessary. An alternative is to have them remember each request as long as possible -- in other words, until they need memory to remember newer requests. Redundant allocation can be reduced in this way, but not completely eliminated.

5.8 SIMULATION

Together with the Fault Tolerant Storage Reclamation project, we have continued development of a fault tolerant resource management simulation begun in fiscal year 1987 under the Future Generation Computer Architectures project. This simulation provides a testbed for our algorithms and ideas. The simulation models resource management (processor and memory management) for a distributed memory, message-passing MIMD system with a very large number of processors. Few assumptions are made about the actual hardware; rather, the physical machine is simulated at an abstract level. The simulation generates statistical information that may be used for analyzing and comparing algorithms.

The simulation permits the user to specify values for various characteristics of the processor fabric as well as the simulated application objects. Some of the characteristics of the processor fabric the user may specify include the dimension of the processor fabric (say, 16 by 16 or 32 by 32 processors), amount of main memory in each individual processor, number and position of secondary storage device controllers, number of processors in each agent's region, number of resource use levels, as well as other characteristics related to the connectivity of the processors, and the communication bandwidth.

For simulated application objects, the user may specify a set of object sizes and associate with each size a relative probability of occurrence of that size object when a new one is created. For example, suppose the user specifies three object sizes 1, 10, and 20 and associates the probabilities 60%, 20% and 10%, respectively. When an object is created, 60% of the time, the object will require one unit of memory for allocation, 20% of the time it will require 10, and 10% of the time it will require 20. The user may specify a set of object creation rates (the percentage of the time an object will create another object when it is accessed) and associate with each rate a relative probability of occurrence of an object with that rate when a new one is created. For example, suppose the user specifies two creation rates: 90% and 0% and associates the probabilities 5% and 95%, respectively. Roughly speaking, this means that 5% of the objects will create another object 90% of the time they are accessed, and 95% will not create another object when they are accessed. Thus, when an object is created 5% of the time a creation rate of 90% will be associated with the object, while 95% of the time a creation rate of 0% will be associated with the object.

As the simulation executes, data indicating how well the system as a whole is functioning are gathered. The user specifies a data collection interval (for example, 200 ticks). At the end of each interval, the number of messages of each type is recorded, as well as information gathered from each processor that includes its resource use data (units of memory in use and free, number of objects, etc.) and the number of, as yet, unprocessed messages in its input queue.

The data gathered can be examined directly or used as input to a graphical display tool. The tool shows placement of agents and superagents in the network and displays an animation that shows the changing processor resource use levels and processor queue lengths as a run of the simulation unfolds.

5.8.1 Initialization

A run of the simulation begins by prompting the user to specify values for various characteristics of the processor fabric as well as the simulated application objects. Default values are provided for many of the required inputs, so that the user need only enter those values that differ from the defaults. After the user is satisfied with the parameter settings, the process of initializing the network begins. The network is created as an array of simulated processor objects, and communication relationships are established. Agents, superagents, and backups are assigned to specific processors in the network. Initialization messages are sent between superagents and their inferiors, between agents and their processors, and between backups and their associated superagents and agents. An initial set of objects, specified by the user, is established in the network. Following initialization, the actual run of the simulation begins.

5.8.2 Two-phase Time Step

The simulation is time driven. There are two phases to each tick of the simulation clock: the computation phase and the communication phase. During the computation phase each processor in the network is given an opportunity to process a number of messages, and service a number of objects which reside in its local memory (this can be thought of as activating a process on the ready queue). To provide for the asynchronous nature of each processor, the processors in the network are accessed randomly.

During the communication phase, (after all processors have been given a turn), the communication of all messages buffered during the computation phase takes place. Messages are transferred between regions in the network. The processor-to-processor communication is simulated at an abstract level and demonstrates two properties: (a) message latency , and (b) contention for communication resources. In the simulation, a group of processors is associated with a *message bag* (figure 5-7).

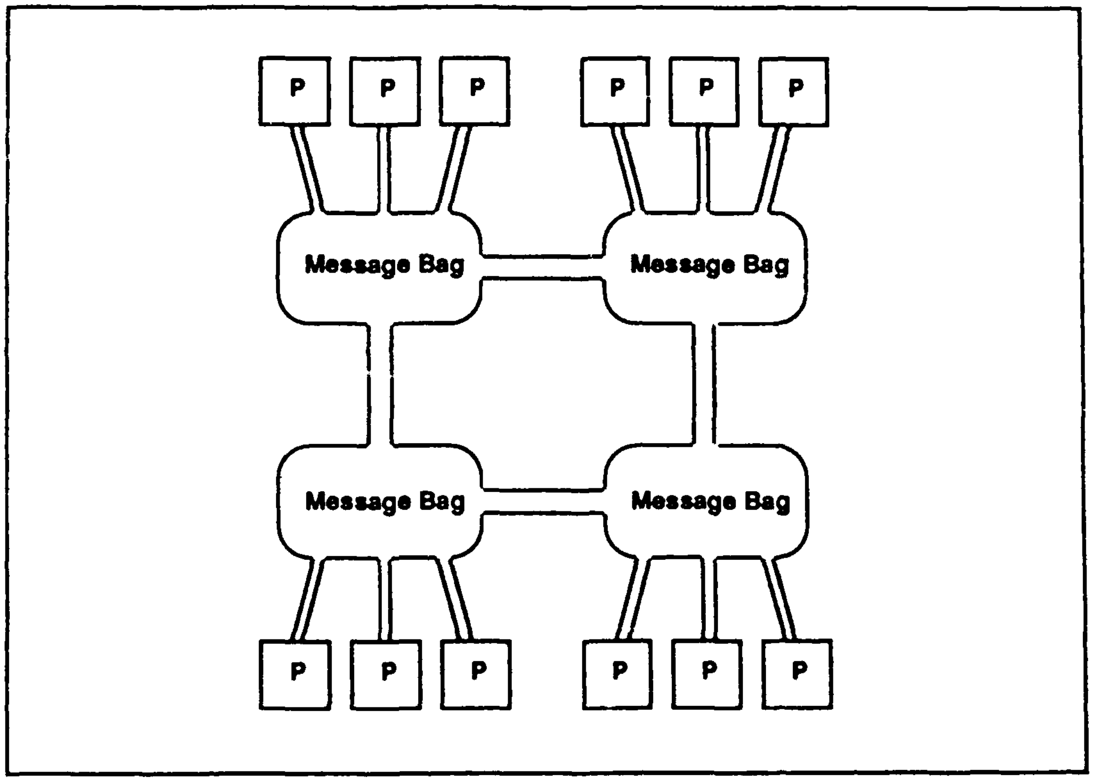


Figure 5-7. Message Bags

When a message is sent from one processor to another, the message is placed in the message bag to which the processor is connected. The message bag accepts all messages it receives and queues them internally during the computation phase. During the communication phase, the message bags are randomly polled and each is given a turn to deliver a message. This polling may be done several times (user-specifiable) before the next computation phase. Messages can be delivered to either the destination processor (to which the bag must be connected), or to another bag (to which the bag is connected). The bag can only deliver a certain number of messages to its processors within each communication phase. In the same way, only a certain number of messages can travel across each connection to another bag during each communication phase.

5.9 SIMULATION LESSONS LEARNED

5.9.1 Redistribution

We found that our redistribution mechanism works well to spread objects throughout the fabric in an equitable manner. Even when the fabric is 75% full, most requests for object allocations can be met by a processor in the same neighborhood as the processor requesting the allocation. The approximate knowledge the superagents have about memory use in their neighborhood is adequate to find the available (free) space in remote areas as the fabric becomes full.

The processor message queues are kept to a reasonable length, usually below ten unprocessed messages at any given time. Even at certain "pressure points" (bottlenecks) in the resource management hierarchy, the queue lengths grow periodically but recover quickly.

5.9.2 Secondary Storage

We found that our strategy for moving objects to secondary storage when primary storage begins to saturate, depends on the rate at which objects are added (either by creation or movement from secondary storage) to primary storage and on the number of secondary storage devices attached to the network. The higher the addition rate, the more secondary storage devices are needed to maintain a reasonably constant memory use level between garbage collection cycles.

Our strategy for using secondary storage in this way does not work well. The number of secondary storage devices that we consider reasonable does not seem to be adequate to maintain a reasonably constant memory use level between garbage collection cycles. Although we have never run our simulation with more than 1024 processors, it is not hard to understand where the difficulty lies.

Let us assume that the time to move an object from a processor to secondary storage is approximately the same as the time needed to add an object to the fabric -- either by creating the object or by moving the object from secondary storage to a processor. To be able to use the secondary storage devices to maintain a system wide memory use level (say, 75%) between garbage collections, means that the number of objects added to the main memory of the system in a given time interval cannot exceed the number of objects moved to secondary storage devices in that time interval. The rate at which objects can be moved to secondary storage depends on the number of secondary storage devices attached to the fabric.

5.9.3 Inexact Information

The number of messages to complete a request for resources must be minimized. The simulation was used to evaluate two different allocation algorithms for resource management, and each algorithm's impact on system performance (number of messages to complete allocation) under the assumption that it is possible for messages to be lost in transit. In one algorithm, the agents have perfect knowledge of each processor's memory use in their region. In the other, the agents have only fuzzy knowledge, that is, estimates of each processor's memory use in their region. To obtain these estimates, whenever a processor crosses a memory boundary (10%, 20%, etc.) it sends a message to its agent reporting its new level.

In both schemes if a request cannot be satisfied in an agent's region, the request is passed up the memory management hierarchy to a superagent. If a request can be satisfied in the agent's region, then in the perfect knowledge scheme, a fault tolerant request always takes six messages to complete (figure 5-8). Three messages are needed to obtain a processor with sufficient memory to hold the new object. In this scheme, agents and allocating processors remember the allocation messages they received and processed previously, so that if the originating processor times out and resends the same request, space will not be allocated a second time. However, agents and allocating processors cannot remember how allocation requests are satisfied forever. One solution is to tell them when they can forget how a given request was satisfied, which requires three additional *forget* messages. In the fuzzy knowledge scheme, a fault tolerant request may take as few as three messages (figure 5-9), but may take more because the agent's knowledge of processor memory use is not perfect. For example, the agent may direct an allocation request to a processor which does not have enough free space to store the object. In that case, the agent may have to try one or more other processors in the region before the request can be satisfied, or before it is decided that the request cannot be satisfied in this agent's region.

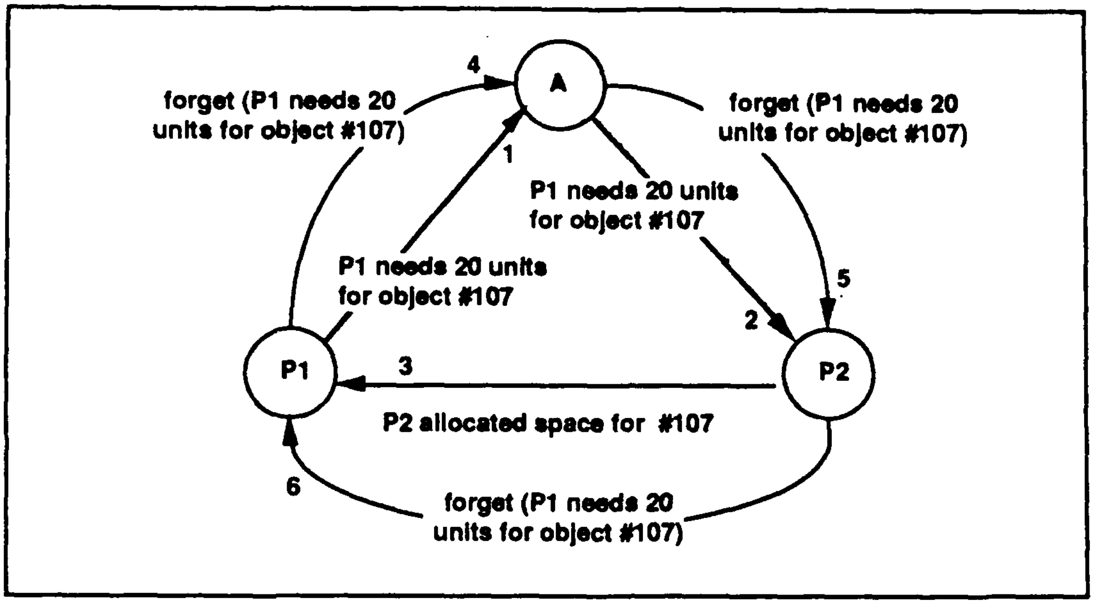

A
P1
P2
1
P1 needs 20 units for object #107
2
P1 needs 20 units for object #107
3
P2 allocated space for #107
4
forget (P1 needs 20 units for object #107)
5
forget (P1 needs 20 units for object #107)
6
forget (P1 needs 20 units for object #107)

Figure 5-8. Allocation Request Perfect Knowledge

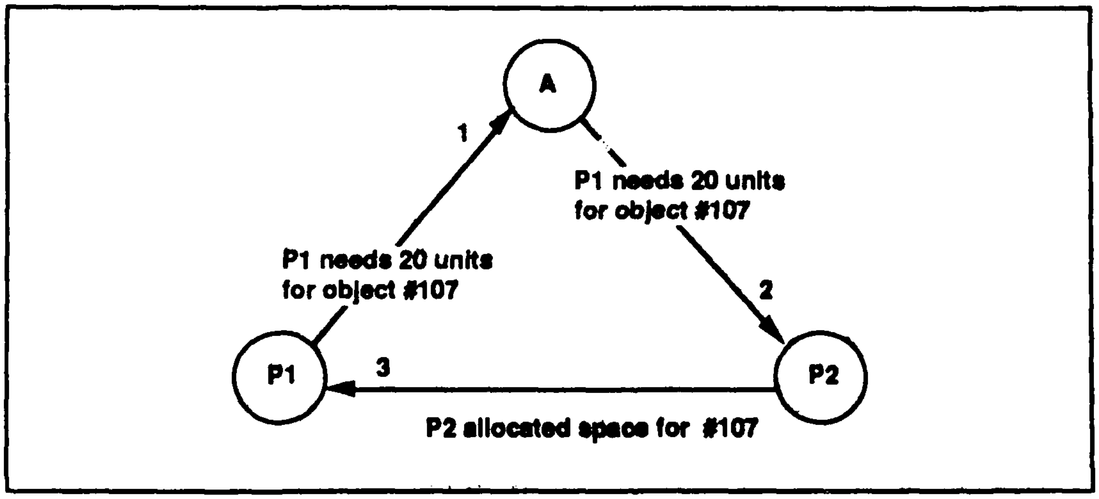

A
P1
P2
1
P1 needs 20 units for object #107
2
P1 needs 20 units for object #107
3
P2 allocated space for #107

Figure 5-9. Allocation Request Fuzzy Knowledge

In evaluating these two algorithms, particular attention was paid to the number of messages sent, number of objects created, and in the fuzzy knowledge scheme, the number of times the agent missed (sent an allocation request to a processor that did not have enough space).

Before fault tolerance and time-outs were implemented, preliminary results showed both algorithms perform about equally when thresholds are set at 10% intervals and there is a low allocation request rate, for example, five percent of the objects request the creation of another object in a given simulation tick. However, when the allocation rate is high, for example, twenty percent of the objects request the creation of another object in a given simulation tick, the fuzzy knowledge scheme has a high miss ratio.

We chose the fuzzy knowledge scheme over the perfect knowledge scheme since it simplifies the fault tolerant implementation of resource management. The perfect knowledge scheme requires more control by the agent and makes fault tolerance more complicated.

SECTION 6

FUTURE WORK

At the present time, the computation manager runs in a simulated multiprocessor environment. Our current implementation of the compiler generates code for branching, looping, and establishing local environments. Next year we plan to implement the computation manager on a distributed-memory, parallel processing architecture and to establish the correctness of the model. In addition, we plan to extend the model of execution and the computation manager to support fault tolerant execution of application programs.

Currently, the resource management simulation includes algorithms for fault tolerant processor and memory management. Before the end of the fiscal year, we plan to include in the simulation an object addressing scheme. Next year we plan to integrate the storage reclamation simulation, developed this year under the Storage Reclamation project (principal investigator T. J. Brando) and the computation manager with the resource management simulation. The integrated simulation will be used to observe the functioning of the operating system as a whole for object oriented programs.

SECTION 7

LIST OF REFERENCES

Agha, G. A., ACTORS: A Model of Concurrent Computation in Distributed Systems, Cambridge, MA: the MIT Press 1986.

Babb, R. G. II, Programming Parallel Processors, Reading, MA: Addison-Wesley, 1988.

Bensley, E. H., Brando, T. J., and Prelle, M. J., "An Execution Model for Distributed Object-Oriented Programming," OOPSLA '88 Conference on Object-Oriented Programming: Systems, Languages and Applications, San Diego, CA, September 1988.

Brando, T. J., Connell, H. E. T., Harris, J. D., and Prelle, M. J., "A Massively Parallel Artificial Intelligence Processor," Proc. Fifth Int. Phoenix Conf. on Computers and Communications, Scottsdale, AZ, March 1986, pp. 638–645.

Brando, T. J., "Distributed, Fault-Tolerant Object Addressing," Proc. Sixth Int. Phoenix Conf. on Computers and Communications, Scottsdale, AZ, February 1987, pp. 242–247.

Buhr, R. J. A., System Design With Ada, Englewood Cliffs, NJ: Prentice-Hall, 1984.

Chow, E., Madan, H., and Peterson, J., "A Real-Time Adaptive Message Routing Network for the Hypercube Computer," Proc. Eighth Real-Time Systems Symposium, IEEE Computer Society, December 1987.

Dally, W. J., and Seitz, C. L., "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," IEEE Transactions on Computers, Vol. C-36, No. 5, May 1987.

Gafni, A., "Space Management and Cancellation Mechanisms for Time Warp," Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, TR-85-341, December 1985.

Halstead, R., "MultiLisp: A Language for Concurrent Symbolic Computation," ACM Trans. on Prog. Languages and Systems, October 1985, pp. 501–538.

Harris, J. D., and Connell, H. E. T., "An Interconnection Scheme for a Tightly Coupled Massively Parallel Computer Network," Proc. Int. Conf. on Computer Design: VLSI in Computers, Port Chester, NY, October 1985, pp. 612–616.

Hwang, K., and Briggs, F. A., Computer Architecture and Parallel Processing, New York: McGraw-Hill, 1984.

Jefferson, D., and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control," Rand Note N-1906AF, the Rand Corporation, Santa Monica, CA, December 1982.

Jefferson, D., "Virtual Time," ACM Trans. on Prog. Languages and Systems, Vol. 7, No. 3, July 1985.

Jefferson, D., et al., "Distributed Simulation and the Time Warp Operating System," Proc. Eleventh ACM Symp. on Operating Systems Principles, Austin, TX, November 1987, pp. 77–93.

Miller, J. S., "MultiScheme: A Parallel Processing System Based on MIT Scheme," MIT/LCS/TR-402, September 1987.

Misra, J., "Distributed Discrete-Event Simulation," Computing Surveys, Vol. 18, No. 1, March 1986, pp. 39–65.

Perrott, R. H., Parallel Programming, Reading, MA: Addison-Wesley, 1987.

Prelle, M. J., "Dynamic Fault-Tolerant Object Addressing," Proc. Sixth Int. Phoenix Conf. on Computers and Communications, Scottsdale, AZ, February 1987, pp. 248–252.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.